# 第8章 驱动模型与总线驱动

Linux内核将设备视为文件，每个设备由一个设备文件表示，内核采用与普通文件相同的接口操作设备，即file\_operations结构体实例。内核为每个设备赋予一个系统内唯一的设备号，保存在设备文件inode中。内核建立了设备驱动数据库，用于管理设备驱动程序（驱动数据结构实例）。

设备文件是特殊文件，在打开设备文件时，将做特殊处理，通过设备号检索设备驱动数据库，找到设备对应的驱动数据结构实例，并将设备驱动数据结构实例中包含的file\_operations实例，赋予file实例，从而可通过此实例操作设备。

内核根据设备与系统之间交换数据的方法，将设备分为字符设备和块设备。字符设备一般只能顺序访问，数据传输量较低，如：鼠标、键盘等。块设备是可以按固定数目字节数（数据块）随机访问的设备，主要是存储设备，如：硬盘、U盘、光盘等。

字符设备、块设备驱动程序在内核设备驱动数据库中由特定的数据结构表示（cdev或gendisk），并分别进行管理。设备驱动程序的主要工作就是定义设备专属的file\_operations实例，定义驱动数据结构实例，并向数据库注册。

系统中的设备不是固定不变的，是可以动态地接入或移出系统，因此设备驱动程序也需要具有动态注册/移出的机制。内核定义了通用驱动模型用于管理内核中的设备和驱动程序（驱动数据结构实例），当向系统接入设备时，驱动模型需要将相应的驱动程序注册进内核，移出设备时将其从设备驱动数据库移除。

通用驱动模型中定义了总线、设备类、设备和驱动等概念，设备用于描述设备硬件信息，硬件信息将传递给驱动，驱动用于匹配设备，匹配成功则注册设备驱动程序、激活设备等。通用驱动模型中的驱动是驱动程序的管理者，负责注册/移除匹配设备的驱动数据结构实例，注意其与驱动程序的区别。

设备和驱动挂接在总线上，在向总线注册设备/驱动时，会触发设备与驱动之间的匹配，匹配成功将完成设备驱动数据结构实例的注册，可理解成加载设备驱动程序到内核。

本章先介绍设备在内核中的表示方法、设备驱动数据库以及通用驱动模型的实现，然后介绍几种常用总线驱动的实现，具体设备驱动程序在后面章节中再做介绍。

## 8.1驱动模型概述

设备驱动程序的主要工作是实现设备驱动对应的cdev或gendisk结构体实例并向内核注册。现代计算机系统中外部设备都通过总线与CPU连接，简单的设备驱动数据库没有解决设备与总线的连接问题及总线驱动的问题。另外，外部设备可以动态地从系统中接入或移出，相应的设备驱动数据结构实例也应该能够动态地添加或移除。

为此，内核在2.6版本引入了通用驱动模型的概念，驱动模型中定义了总线、设备类、设备、驱动等概念，用于管理系统中的实际总线、硬件设备和驱动程序（设备驱动数据结构实例）。总线管理着其下设备和驱动，添加设备或驱动时，会触发两者之间的匹配，匹配成功，将由驱动注册设备驱动数据结构实例，并激活设备。

### 8.1.1组织结构

通用驱动模型中定义了总线bus\_type、设备类class、设备device和驱动device\_driver等数据结构。总线bus\_type表示系统中实际的总线（也可以是虚拟总线，芯片内部总线等），如：USB总线、SPI总线等，设备类class用于管理同一类型的设备，device表示设备的硬件信息，device\_driver表示驱动（用于管理设备驱动程序）。下图示意了字符设备的通用驱动模型框架，块设备也类似，只不过它也有专门的设备驱动数据库。

同一条总线上的设备device实例挂接到对应总线bus\_type实例的设备链表上，相应的驱动device\_driver实例挂接在总线的驱动链表上。当向总线注册设备或驱动时（注册设备时可能动态创建设备文件），将触发总线的匹配操作，寻找匹配的驱动或设备（扫描驱动或设备链表）。若匹配成功则调用总线bus\_type实例（或驱动device\_driver实例）中定义的probe()函数，probe()函数完成设备驱动数据结构实例的定义和注册。进程可通过设备文件操作设备，设备文件中包含设备号，依此找到设备驱动数据结构实例，从而实现对设备的操作。



### 8.1.2初始化

内核通过kobject结构体跟踪和管理驱动模型中的各数据结构实例，并导出到sysfs文件系统。在初始化函数中将创建各目录项，用于管理驱动模型中的数据结构实例，如下图所示。



以上目录项在驱动模型初始化函数driver\_init()中创建，函数调用关系如下：

kernel\_init->kernel\_init\_freeable()->do\_basic\_setup()->driver\_init()。

driver\_init()函数定义在/drivers/base/init.c文件内，代码如下：

void \_\_init driver\_init(void)

{

**devtmpfs\_init()**; /\*注册devtmpfs文件系统类型，创建内核线程，/drivers/base/devtmpfs.c\*/

devices\_init(); /\*设备初始化，/drivers/base/core.c\*/

buses\_init(); /\*总线初始化，/drivers/base/bus.c\*/

classes\_init(); /\*设备类初始化，/drivers/base/class.c\*/

firmware\_init(); /\*固件初始化，/drivers/base/firmware.c\*/

hypervisor\_init(); /\*/drivers/base/hypervisor.c\*/

platform\_bus\_init(); /\*平台总线初始化，/drivers/base/platform.c\*/

cpu\_dev\_init(); /\*注册cpu\_subsys总线，挂载cpu设备到此总线，/drivers/base/cpu.c\*/

memory\_dev\_init(); /\*注册memory\_subsys总线和内存设备，/drivers/base/memory.c\*/

container\_dev\_init(); /\*注册container\_subsys总线，/drivers/base/container.c\*/

of\_core\_init(); /\*设备树初始化，见下文，/drivers/of/base.c\*/

}

以上初始化函数简介如下：

**●devtmpfs\_init()：**在/drivers/base/devtmpfs.c文件内实现，需选择DEVTMPFS配置选项，否则为空操作。函数完成devtmpfs文件系统类型的注册以及创建设备文件守护线程kdevtmpfs的创建和启动，此线程用于在注册device实例时自动创建设备文件。

**●devices\_init()：**在/drivers/base/core.c文件内实现，代码详见本章下文。函数在sysfs文件系统中创建以下目录：

/sys/devices/：导出驱动模型中device实例。

/sys/dev/block/：导出块设备，其下文件名称为“主设备号：从设备号”，是到/sys/devices/下设备的符号链接。

/sys/dev/char/：导出字符设备，其下文件名称为“主设备号：从设备号”，是到/sys/devices/下设备的符号链接。

**●buses\_init()：**总线初始化函数，定义在/drivers/base/bus.c文件内，详见本章下文。函数在sysfs文件系统内创建以下目录：/sys/bus/，/sys/devices/system，前者用于导出注册的总线。

**●classes\_init()：**设备类初始化函数，在/drivers/base/class.c文件内实现，在sysfs文件系统中创建目录/sys/class/，用于导出注册的设备类，详见本章下文。

**●firmware\_init()：**在/drivers/base/firmware.c文件内实现，代码如下：

int \_\_init firmware\_init(void)

{

firmware\_kobj = kobject\_create\_and\_add("firmware", NULL);

...

return 0;

}

函数在sysfs文件系统中创建目录/sys/fireware，用于导出固件信息。

**●hypervisor\_init()：**在/drivers/base/hypervisor.c文件内实现，代码如下：

int \_\_init hypervisor\_init(void)

{

hypervisor\_kobj = kobject\_create\_and\_add("hypervisor", NULL);

...

return 0;

}

函数在sysfs文件系统中创建目录/sys/hypervisor。

**●platform\_bus\_init()：**平台platform总线初始化函数，在/drivers/base/platform.c文件内实现，主要完成平台总线的注册，详见本章下文。

**● cpu\_dev\_init()：**在/drivers/base/cpu.c文件内实现，代码如下：

void \_\_init cpu\_dev\_init(void)

{

if (subsys\_system\_register(&**cpu\_subsys**, cpu\_root\_attr\_groups)) /\*注册子系统，/drivers/base/bus.c\*/

panic("Failed to register CPU subsystem");

cpu\_dev\_register\_generic(); /\*挂载cpu设备至cpu\_subsys总线，/drivers/base/cpu.c\*/

}

cpu\_dev\_init()函数完成cpu\_subsys总线的注册和cpu设备的挂载。

注册子系统的函数在/drivers/base/bus.c文件内实现，它先注册总线，再将总线以设备形式注册到内核。

若选择了GENERIC\_CPU\_DEVICES配置选项，CPU将会当作设备（由cpu结构体表示）挂载到总线cpu\_subsys上。

**●memory\_dev\_init()：**函数在/drivers/base/memory.c文件内实现，主要完成内存总线类型memory\_subsys及内存设备的注册。

**●container\_dev\_init()：**在/drivers/base/container.c文件内实现，主要完成container\_subsys总线（子系统）的注册。

**●of\_core\_init()：**在/drivers/of/base.c文件内实现，需选择OF配置选项，表示内核支持设备树，否则为空操作。函数内主要工作是创建/sys/fireware/devicetree/目录，并将设备树节点导出到此目录下。

## 8.2设备管理

在介绍通用驱模型前，先介绍一下设备在内核中的表示（设备文件），以及内核对设备驱动程序的管理（设备驱动数据库）。

设备在Linux系统中由设备文件表示，设备文件保存的主要信息有设备类型和设备号等。设备类型即字符设备或块设备，设备号用于在打开设备文件时在设备驱动数据库中搜索设备驱动。

### 8.2.1设备号

设备号由主设备号和从设备号组成，字符设备与块设备分开编号，也就是说这两种类型设备可以使用相同的设备号，内核通过设备类型进行区分。一般同一类型的设备具有相同的主设备号，通过从设备号来区分具体的设备。

最初设备号由16位整数表示，高8位表示主设备号，低8位表示从设备号。后因设备增多设备号不够用，因此将设备号扩展为32位，高12位表示主设备号，低20位表示从设备号。

保存在旧的文件系统（介质文件系统）中的设备文件，其节点中保存的设备号只使用了16个比特位，为与旧的布局相兼容，在设备号需要在外部表示时（如：外部文件系统中的设备文件）则采用外部表示方法。如下图所示，外部表示方式中0~7位及20~31位合并表示从设备号，而8~19位表示主设备号。



具体设备的设备号在/Documentation/devices.txt文件内定义，内核在/include/uapi/linux/major.h头文件内定义了表示主设备号的宏。

设备号在内核内部由dev\_t数据类型表示（32位无符号整数），内核在/include/linux/kdev\_t.h头文件中定义了以上设备号表示方式之间的转换函数，例如：

#define MINORBITS 20

#define MINORMASK ((1U << MINORBITS) - 1)

#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS)) /\*获取dev\_t中主设备号\*/

#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK)) /\*获取dev\_t中从设备号\*/

#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi)) /\*主、从设备号合成dev\_t实例\*/

设备文件中保存了设备号和设备类型，在打开设备文件时，内核通过设备号和设备类型查找设备驱动数据库。在以前的Linux系统中，设备文件静态保存在外部文件系统中（需要用户创建），因此需要保证驱动程序注册的设备号与设备文件中的设备号相同，否则打开设备文件时将无法找到正确的设备驱动数据结构。

后来Linux系统引入了udev（mdev）机制，在向内核添加设备时（注意是添加设备，添加驱动时不行）将触发uevent事件，通过环境变量向用户空间传递设备信息，内含设备号和类型信息，然后由用户守护进程udev根据设备信息自动创建设备文件。

现在Linux内核引入了更智能化和简单化的方法，即内核在调用device\_add(dev)函数添加device（设备）实例时，如果device实例设置了设备号，内核将自动唤醒内核守护线程在devtmpfs文件系统（/dev/）中自动创建设备文件（见本章下文）。驱动中设备号与设备文件中设备号由内核自动实现统一，驱动程序和用户进程只需关注设备文件名称，而不再需要将某个设备固定赋予某个设备号。

### 8.2.2驱动数据库

在设备驱动数据库中，字符设备由cdev结构体表示，块设备由gendisk结构体表示，这两个结构体的定义在介绍字符设备和块设备驱动程序时再做介绍。内核对字符设备和块设备驱动数据结构实例的管理采用了相同的数据结构，称之为设备驱动数据库。

#### 1数据结构

内核定义了kobj\_map结构体用于管理设备驱动数据结构实例，结构体定义在/drivers/base/map.c文件内：

struct kobj\_map {

struct probe {

struct probe \*next; /\*指向下一个probe实例，构成单链表\*/

dev\_t **dev**; /\*设备号，包含起始从设备号\*/

unsigned long  **range**; /\*从设备号数量\*/

struct module \*owner; /\*模块指针\*/

kobj\_probe\_t **\*get**; /\*获取设备驱动数据结构中kobject结构体成员指针函数\*/

int (\*lock) (dev\_t, void \*);

void \***data;**  /\*指向cdev或gendidk结构体\*/

} \*probes[255]; /\*probe结构指针数组\*/

struct mutex \*lock; /\*互斥量\*/

};

kobj\_map结构体实际上是一个probe结构体指针数组，数组项数为255。probe结构体中各成员定义如下：

●**next**：指向下一个probe实例，构成单链表。

**●dev**：probe实例所适配设备的设备号，包含主设备号和起始从设备号。

●**range**：从设备号数量，从设备号取值范围为[MINORS(dev)，MINORS(dev)+range-1]。probe实例管理的cdev或gendisk实例可适用于同一主设备号的多个从设备，即多个从设备共用一个驱动程序。

●**owner**：模块指针。

●**get**：kobj\_probe\_t类型函数指针，函数返回设备驱动数据结构中kobject结构体成员指针。通常kobject结构内嵌在被跟踪数据结构内部，获取kobject指针后，通过容器机制可获取被跟踪数据结构实例指针。

kobj\_probe\_t函数类型定义在/include/linux/kobj\_map.h头文件内：

**typedef struct kobject \*kobj\_probe\_t(dev\_t, int \*, void \*);**

**●data**：指向设备驱动数据结构实例，指向**cdev**或**gendisk**结构体，data将作为kobj\_probe\_t(dev\_t, int \*, void \*)函数的void指针参数。

每个probe结构实例管理的设备驱动数据结构实例（cdev或gendisk），可适用于同一主设备号下的多个从设备。一个主设备号也可对应多个probe实例表示，每个实例适用一段范围的从设备号。

probe实例由所适用设备的主设备号确定添加到probe指针数组中哪一项，即probe实例关联的数组项为probes[MAJOR(dev)%255]。同一指针数组项下的probe实例通过next指针成员组成单链表。

内核为字符设备和块设备分别创建了kobj\_map实例，用于管理设备驱动数据结构实例，称之为设备数据库，如下图所示。



#### 2驱动数据库操作

内核在初始化阶段为字符设备和块设备分别创建了kobj\_map结构体实例，注册设备驱动时需要向设备驱动数据库添加probe实例，实例data成员指向cdev或gendisk实例。在打开设备文件时，打开操作函数根据设备类型和设备号搜索设备驱动数据库，找到设备对应的驱动数据结构实例。

##### **■初始化**

kobj\_map\_init()函数用于创建和初始化kobj\_map结构体实例，函数代码如下（/drivers/base/map.c）：

struct kobj\_map \*kobj\_map\_init(kobj\_probe\_t \***base\_probe**, struct mutex \*lock)

/\*base\_probe：kobj\_probe\_t函数指针\*/

{

struct kobj\_map \*p = **kmalloc(sizeof(struct kobj\_map), GFP\_KERNEL)**; /\*创建kobj\_map实例\*/

struct probe \*base = **kzalloc(sizeof(\*base), GFP\_KERNEL)**; /\*创建一个初始probe实例\*/

int i;

if ((p == NULL) || (base == NULL)) {

...

}

**base->dev = 1;**  /\*主设备号为0，从设备号为1\*/

**base->range = ~0;** /\*从设备号范围是无符号整数最大值\*/

**base->get = base\_probe**; /\*函数指针，由参数传递\*/

for (i = 0; i < 255; i++)

**p->probes[i] = base**; /\*所有指针数组项指向初始probe实例\*/

**p->lock = lock;**

return p; /\*返回kobj\_map实例指针\*/

}

kobj\_map\_init()函数比较简单，需要向函数传递kobj\_probe\_t函数指针和互斥量指针参数。函数内创建一个kobj\_map和probe结构体实例，并对实例进行初始化，最后返回kobj\_map实例指针。新创建kobj\_map实例内probes[]指针数组项都指向初始的probe实例，如下图所示：



##### **■添加设备驱动**

向设备驱动数据库添加新项时将调用kobj\_map()函数，函数定义如下（/drivers/base/map.c）：

int kobj\_map(struct kobj\_map \*domain, dev\_t dev, unsigned long range,struct module \*module, \

kobj\_probe\_t \*probe,int (\*lock)(dev\_t, void \*), void **\*data**)

/\*

\* domain：kobj\_map实例指针，**dev：设备号（含起始从设备号），range：从设备号数量**，

\* probe：kobj\_probe\_t函数指针，**data：cdev或gendisk实例指针**，module：模块指针。

\*/

{

unsigned n = MAJOR(dev + range - 1) - MAJOR(dev) + 1; /\*需创建probe实例数量\*/

unsigned **index = MAJOR(dev)**; /\*主设备号\*/

unsigned i;

struct probe \*p;

if (n > 255)

n = 255;

**p = kmalloc\_array(n, sizeof(struct probe), GFP\_KERNEL)**; /\*创建probe结构实例数组\*/

if (p == NULL)

return -ENOMEM;

for (i = 0; i < n; i++, p++) { /\*初始化每个probe实例，每个实例成员值相同\*/

p->owner = module;

**p->get = probe**; /\*获取kobject函数指针\*/

**p->lock = lock**;

**p->dev = dev;** /\*设备号\*/

**p->range = range;** /\*从设备号数量\*/

**p->data = data;**  /\*指向cdev或gendisk实例\*/

}

mutex\_lock(domain->lock);

for (i = 0, p -= n; i < n; i++, p++, index++) { /\*由主设备号将probe实例添加到kobj\_map中\*/

**struct probe \*\*s = &domain->probes[index % 255];**  /\*指针数组项，每次循环index加1\*/

while (\*s && (\*s)->range < range) /\*probe实例在单链表中按range值从小到大排序\*/

s = &(\*s)->next;

p->next = \*s;

\*s = p;

}

mutex\_unlock(domain->lock);

return 0; /\*成功返回0\*/

}

kobj\_map()函数比较简单，主要是创建probe实例并初始化，由主设备号确定将实例添加到kobj\_map中指针数组中的哪一项，实例以range值从小到大，在单链表中从左至右排序。需要注意的是，如果起始从设备号加上从设备号数量大于20位从设备号能表示的最大值，则需要创建多个probe实例，每个probe实例各成员值是一样的，但是添加到不同的probe指针数组项中。

设备驱动程序的主要工作就是构建设备驱动数据结构（cdev或gendisk）实例，在通用驱动模型中驱动的probe()函数中将其添加到设备驱动数据库。

##### **■查找设备驱动**

在打开设备文件时，需要根据设备文件中保存的设备号，查找设备驱动数据库获取设备驱动数据结构实例，查找函数返回跟踪cdev或gendisk实例的kobject实例指针，调用者可通过容器等机制获取cdev或gendisk实例指针。

查找设备数据库的kobj\_lookup()函数定义如下（/drivers/base/map.c）：

struct kobject \*kobj\_lookup(struct kobj\_map \*domain, dev\_t dev, int \*index)

/\*

\*domain：kobj\_map实例指针，dev：查找设备号，含主从设备号，

\*index：\*index保存查找从设备号基于probe实例起始从设备号的偏移量。

\*/

{

struct kobject \*kobj;

struct probe \*p;

unsigned long best = ~0UL;

retry:

mutex\_lock(domain->lock);

for (p = domain->**probes[MAJOR(dev) % 255]**; p; p = p->next) { /\*遍历probe单链表\*/

struct kobject \*(\*probe)(dev\_t, int \*, void \*);

struct module \*owner;

void \*data;

if (p->dev > dev || p->dev + p->range - 1 < dev) /\*查找设备号不在probe表示的范围内\*/

continue;

if (p->range - 1 >= best)

break;

if (!try\_module\_get(p->owner))

continue;

/\*找到含盖dev\_t设备号的probe实例\*/

owner = p->owner;

**data = p->data**; /\*cdev或gendisk实例指针\*/

**probe = p->get;**

best = p->range - 1;

**\*index = dev - p->dev**; /\*从设备号相对于p->dev起始从设备号的偏移量\*/

if (p->lock && p->lock(dev, data) < 0) {

module\_put(owner);

continue;

}

mutex\_unlock(domain->lock);

**kobj = probe(dev, index, data**); /\*调用p->get()函数获取kobject实例指针\*/

module\_put(owner);

if (kobj)

**return kobj**; /\*返回kobject实例指针\*/

goto retry;

}

mutex\_unlock(domain->lock);

return NULL;

}

查找函数根据设备号dev中的主设备号索引kobj\_map指针数组，扫描probes[MAJOR(dev) % 255]表项中的probe实例链表，找到含盖dev设备号的probe实例，获取从设备号偏移量后，调用p->get()函数获取跟踪设备驱动数据结构实例的kobject实例，并返回kobject实例指针。

\*index保存参数dev中从设备号相对于probe实例中的起始从设备号的偏移量。例如，假设参数dev的从设备号为2，而probe实例表示的从设备号范围为[0，12]，则\*index为2，若从设备号范围是[1，12]，则\*index为1。

void kobj\_unmap(struct kobj\_map \*domain, dev\_t dev, unsigned long range)函数负责从设备驱动数据库中删除设备对应的probe结构实例，源代码请读者自行阅读。

### 8.2.3创建设备文件

设备文件是进程操作设备的接口，曾经设备文件需要静态保存在外部根文件系统/dev/目录下，后来使用了udev机制，在向内核注册设备时通过uevent机制向用户空间发送信息，由用户守护进程udev创建设备文件。现在内核采用了更简单的方法，内核在启动阶段将devtmpfs文件系统（ramfs或tmpfs实例）挂载到/dev/目录下，并创建内核守护线程。在向内核添加设备时，由守护线程在/dev/目录下自动创建设备文件。

使用devtmpfs需选择DEVTMPFS配置选项，devtmpfs文件系统是ramfs或tmpfs文件系统实例，在添加设备device实例时创建设备文件，系统关机时消失。因为自动创建的设备文件其设备号由内核在设备device实例中提取，可保证两个设备号是相同的，因此驱动程序并不需要特别在意设备使用哪个设备号，用户进程只是通过设备文件名称识别设备。

#### 1 devtmpfs文件系统

devtmpfs文件系统通常挂载到/dev/目录下，用于存放设备文件，devtmpfs是ramfs或tmpfs实例，内核保证只有一个devtmpfs文件系统挂载到根文件系统，以保证设备文件的唯一性。

devtmpfs文件系统类型定义在/drivers/base/devtmpfs.c文件内：

static struct file\_system\_type dev\_fs\_type = {

.name = "devtmpfs", /\*文件类型名称\*/

.mount = **dev\_mount**, /\*挂载函数\*/

.kill\_sb = kill\_litter\_super,

};

devtmpfs文件系统类型的挂载函数定义如下（/drivers/base/devtmpfs.c）：

static struct dentry \*dev\_mount(struct file\_system\_type \*fs\_type, int flags,const char \*dev\_name, \

void \*data)

{

#ifdef CONFIG\_TMPFS

return **mount\_single(fs\_type, flags, data, shmem\_fill\_super)**; /\*套用tmpfs文件系统类型\*/

#else

return **mount\_single(fs\_type, flags, data, ramfs\_fill\_super)**; /\*套用ramfs文件系统类型\*/

#endif

}

mount\_single()函数保证挂载的文件系统在内核中只有一个实例（超级块实例），即使进行多次挂载，各挂载点挂载的也是同一个文件系统实例。

devtmpfs文件系统初始化流程如下图所示：



devtmpfs\_init()函数向内核注册devtmpfs文件系统类型，并创建kdevtmpfs内核守护线程，用于自动创建设备文件。

在创建kernel\_init线程之前内核已经挂载了rootfs作为初始的根文件系统。kdevtmpfs守护线程首次运行时，将为线程创建新的挂载命名空间，并将devtmpfs文件系统挂载到根文件系统根目录项，并设devtmpfs根目录项为守护线程根目录和当前工作目录。kdevtmpfs线程创建设备文件时，在其当前工作目录下创建。

由于kdevtmpfs线程是在新挂载命名空间中挂载devtmpfs，因此挂载只对守护线程可见，对其它线程不可见。

如果内核挂载了外部块设备作为根文件系统，prepare\_namespace()函数中调用**devtmpfs\_mount("dev")**函数，可能自动挂载devtmpfs至/dev/目录。自动挂载的条件是（满足其一即可）：

●选择了DEVTMPFS\_MOUNT配置选项。

●传递了“devtmpfs.mount=1”命令行参数。

如果内核没有挂载外部块设备作为根文件系统（仍使用rootfs）或传递了“devtmpfs.mount=0”命令行参数（不管有没有选择DEVTMPFS\_MOUNT配置选项），devtmpfs\_mount()函数中将不会挂载devtmpfs，需要用户在启动后手动挂载。

devtmpfs自动挂载结果如下图所示：



kdevtmpfs线程创建设备文件时，在其当前工作目录下创建，即在devtmpfs根目录下创建。由于devtmpfs同时挂载到/dev/目录下（同一个文件系统实例），因此其对内核根文件系统也可见。

初始化函数devtmpfs\_init()和devtmpfs\_mount()定义如下（/drivers/base/devtmpfs.c）：

int \_\_init devtmpfs\_init(void)

{

int err = **register\_filesystem(&dev\_fs\_type)**; /\*注册devtmpfs文件系统类型\*/

...

**thread = kthread\_run(devtmpfsd, &err, "kdevtmpfs")**;

/\*创建内核守护线程kdevtmpfs，线程执行函数为devtmpfsd()\*/

if (!IS\_ERR(thread)) {

**wait\_for\_completion(&setup\_done)**;

/\*kernel\_init线程进入睡眠，等待kdevtmpfs线程将其唤醒\*/

} else {

...

}

...

return 0;

}

devtmpfs\_init()函数由kernel\_init内核线程调用，函数内注册devtmpfs文件系统类型，并创建kdevtmpfs内核守护线程，然后kernel\_init线程进入睡眠等待，等待kdevtmpfs守护线程首次运行时将其唤醒。

devtmpfs\_mount()函数由prepare\_namespace()函数调用，负责将devtmpfs挂载到/dev/目录，函数定义如下：

int devtmpfs\_mount(const char \*mntdir)

/\*mntdir：挂载目录为"dev"，基于当前工作目录（/root/）\*/

{

int err;

if (!**mount\_dev**) /\*是否自动挂载\*/

return 0;

if (!thread) /\*kdevtmpfs守护线程指针\*/

return 0;

err = **sys\_mount**("devtmpfs", (char \*)mntdir, "devtmpfs", MS\_SILENT, NULL); /\*挂载\*/

if (err) /\*输出信息\*/

printk(KERN\_INFO "devtmpfs: error mounting %i\n", err);

else

printk(KERN\_INFO "devtmpfs: mounted\n");

return err;

}

#### 2守护线程

kdevtmpfs内核守护线程执行函数devtmpfsd()定义如下（/drivers/base/devtmpfs.c）：

static int devtmpfsd(void \*p)

{

char **options[] = "mode=0755"**;

int \*err = p;

**\*err = sys\_unshare(CLONE\_NEWNS)**; /\*为守护线程创建新挂载命名空间\*/

if (\*err)

goto out;

**\*err = sys\_mount("devtmpfs", "/", "devtmpfs", MS\_SILENT, options);**

/\*首次运行将devtmpfs挂载到rootfs文件系统根目录项\*/

/\*在新挂载命名空间中的挂载，对kernel\_init线程不可见\*/

if (\*err)

goto out;

**sys\_chdir("/..")**; /\*设置kdevtmpfs线程当前工作目录和根目录为devtmpfs根目录\*/

**sys\_chroot(".")**;

**complete(&setup\_done)**; /\*首次运行至此，唤醒kernel\_init线程\*/

/\*线程进入无限循环，用于创建设备文件（详见下文）\*/

while (1) {

spin\_lock(&req\_lock);

while (requests) { /\*扫描requests指向的请求req实例链表，逐个为req实例创建设备文件\*/

struct req \*req = requests;

**requests = NULL**;

spin\_unlock(&req\_lock);

while (req) {

struct req \*next = req->next;

**req->err = handle(req->name, req->mode,req->uid, req->gid, req->dev)**;

/\*调用函数handle()创建设备文件，/drivers/base/devtmpfs.c\*/

**complete(&req->done)**; /\*唤醒在请求上等待的进程\*/

**req = next**;

}

spin\_lock(&req\_lock);

}

/\*requests链表为空，守护线程睡眠等待\*/

\_\_set\_current\_state(TASK\_INTERRUPTIBLE); /\*线程进入睡眠\*/

spin\_unlock(&req\_lock);

**schedule()**; /\*进程调度，唤醒后继续循环\*/

}

return 0;

out:

complete(&setup\_done);

return \*err;

}

kdevtmpfs线程首次运行将创建新挂载命名空间，将devtmpfs文件系统挂载至rootfs文件系统根目录。kdevtmpfs线程的根目录和当前工作目录都指向devtmpfs文件系统根目录。

随后，kdevtmpfs线程进入无限循环，在循环中扫描requests指向的请求req实例链表，逐个为req实例创建设备文件，并唤醒在请求上睡眠的进程，requests链表为空时，线程睡眠。

#### 3创建设备文件

内核需要创建设备文件时，需要构建请求req结构体实例，添加到requests链表，唤醒kdevtmpfs守护线程。kdevtmpfs线程扫描requests链表，根据请求req实例信息在devtmpfs根目录下创建设备文件。

请求req结构体定义在/drivers/base/devtmpfs.c文件内：

static struct req {

struct req \*next; /\*下一个请求，请求组成单链表\*/

struct completion **done**; /\*完成量，等待设备文件创建完成，由kdevtmpfs内核线程唤醒\*/

int err; /\*创建设备文件返回值\*/

const char \***name**; /\*设备文件名称，如：hd1\*/

umode\_t **mode**; /\*文件模式， 0 表示删除设备文件\*/

kuid\_t uid;

kgid\_t gid;

struct device **\*dev**; /\*设备device实例指针，device结构体见下文\*/

} \*requests;

requests是一个全局变量，指向req实例单链表，向kdevtmpfs内核线程传递请求。

在向内核注册表示设备的device实例时，将调用**devtmpfs\_create\_node**(struct device \*dev)函数触发设备文件的创建（device实例中赋予了主设备号），此函数内将构建请求req实例，将其添加到requests指向的单链表，唤醒kdevtmpfs内核守护线程，由守护线程扫描单链表逐个创建设备文件，如下图所示。



**devtmpfs\_create\_node()**函数定义如下（/drivers/base/devtmpfs.c）：

int devtmpfs\_create\_node(struct device \*dev)

{

const char \*tmp = NULL;

**struct req req**; /\*请求req实例\*/

if (!thread)

return 0;

**req.mode = 0**;

req.uid = GLOBAL\_ROOT\_UID; /\*uid，gid设为0\*/

req.gid = GLOBAL\_ROOT\_GID;

**req.name = device\_get\_devnode(dev, &req.mode, &req.uid, &req.gid, &tmp)**;

/\*获取设备文件名称，/drivers/base/core.c\*/

if (!req.name)

return -ENOMEM;

if (req.mode == 0) /\*如果mode为0\*/

req.mode = 0600; /\*设备文件访问权限，默认只有文件主有读写权限\*/

if (is\_blockdev(dev)) /\*dev->class == &block\_class??\*/

req.mode |= S\_IFBLK; /\*块设备文件，设置文件类型\*/

else

req.mode |= S\_IFCHR; /\*字符设备文件，设置文件类型\*/

**req.dev = dev**; /\*device实例\*/

init\_completion(&req.done);

spin\_lock(&req\_lock);

**req.next = requests**;

**requests = &req**; /\*req添加到requests单链表头部\*/

spin\_unlock(&req\_lock);

**wake\_up\_process(thread)**; /\*唤醒kdevtmpfs内核线程\*/

**wait\_for\_completion(&req.done)**; /\*等待设备文件创建完成时，唤醒本进程\*/

kfree(tmp);

return **req.err**;

}

devtmpfs\_create\_node()函数构建req结构体实例，将其添加到requests单链表的头部，然后唤醒守护线程kdevtmpfs，当前进程在req.done完成量上睡眠等待，当kdevtmpfs守护线程创建完设备文件后，将唤醒在req.done上睡眠等待的进程。

kdevtmpfs线程被唤醒后，如果requests链表不为空，则为链表中每个请求req实例创建设备文件，直至链表为空线程进入睡眠。

kdevtmpfs线程中创建设备文件的handler()函数定义如下：

static int handle(const char \*name, umode\_t mode, kuid\_t uid, kgid\_t gid,struct device \*dev)

/\*name：设备文件名称，mode：文件模式，dev：device实例指针，内含设备号\*/

{

if (mode) /\*非0，创建设备文件\*/

return **handle\_create(name, mode, uid, gid, dev)**; /\*创建设备文件，/drivers/base/devtmpfs.c\*/

else /\*为0，删除设备文件\*/

return handle\_remove(name, dev); /\*删除设备文件\*/

}

handle\_create()函数根据设备名称、设备号、访问权限等信息，调用devtmpfs文件系统inode\_operations结构实例中的mknod()函数（ramfs\_mknod()）创建设备文件对应的dentry和inode实例。对于ramfs/tmpfs文件系统，创建设备节点的操作就是直接创建dentry和inode实例，文件系统没有后备存储设备。

在创建设备文件时，文件名至关重要，因为用户进程只能通过设备文件访问设备，设备号由device实例传递。设备文件名由device\_get\_devnode()函数获取，函数定义如下（/drivers/base/core.c），读者可学习完下面的驱动模型相关知识后再回过头来阅读此函数。

const char \*device\_get\_devnode(struct device \*dev,umode\_t \*mode, kuid\_t \*uid, kgid\_t \*gid, \

const char \*\*tmp)

{

char \*s;

\*tmp = NULL;

/\*优先调用设备类型device\_type实例提供的设备文件名\*/

if (dev->type && dev->type->devnode)

**\*tmp = dev->type->devnode(dev, mode, uid, gid)**;

if (\*tmp)

return \*tmp; /\*tmp不为NULL，返回\*/

/\*tmp为NULL，继续往下执行，设备类class可能提供设备文件名\*/

if (dev->class && dev->class->devnode)

\*tmp = **dev->class->devnode(dev, mode)**;

if (\*tmp)

return \*tmp; /\*tmp不为NULL，返回\*/

/\*如果设备类型和设备类没有提供设备名称，则通过dev\_name()获取设备名称\*/

if (**strchr(dev\_name(dev), '!') == NULL**) /\*名称字符串中如果不包含‘!’字符\*/

return **dev\_name(dev)**; /\*返回设备名称，dev->init\_name 或dev->kobject名称\*/

/\*如果名称字符串中包含 '!' 则用 '/' 代替\*/

s = kstrdup(dev\_name(dev), GFP\_KERNEL);

if (!s)

return NULL;

strreplace(s, '!', '/');

return **\*tmp = s;**  /\*dev->init\_name 或dev->kobject名称\*/

}

删除设备文件的接口函数为**devtmpfs\_delete\_node**(struct device \*dev)，删除设备文件也是由kdevtmpfs守护线程完成（req.mode=0），源代码请读者自行阅读。

### 8.2.4设备文件操作

设备文件是特殊文件，VFS在打开文件时，当发现打开的是特殊文件时，将调用init\_special\_inode(inode, mode, dev)函数对文件inode实例进行初始化。

init\_special\_inode()函数定义如下（/fs/inode.c）：

void init\_special\_inode(struct inode \*inode, umode\_t mode, dev\_t rdev)

/\*inode：inode实例指针，mode：文件访问模式，rdev：设备号\*/

{

inode->i\_mode = mode;

if (**S\_ISCHR(mode)**) { /\*是否是字符设备\*/

inode->i\_fop = &**def\_chr\_fops**; /\*字符设备文件操作结构实例，/fs/char\_dev.c\*/

inode->i\_rdev = **rdev**; /\*设备号\*/

} else if (**S\_ISBLK(mode)**) { /\*块设备文件\*/

inode->i\_fop = &**def\_blk\_fops**; /\*块设备文件操作结构实例，/fs/block\_dev.c\*/

inode->i\_rdev = **rdev**; /\*设备号\*/

} else if (S\_ISFIFO(mode)) /\*命名管道\*/

inode->i\_fop = &pipefifo\_fops;

else if (S\_ISSOCK(mode)) /\*套接字\*/

;

else

... /\*输出信息\*/

}

字符设备文件默认的文件操作结构实例定义如下（/fs/char\_dev.c）：

const struct file\_operations **def\_chr\_fops** = {

.open = **chrdev\_open**, /\*打开函数\*/

.llseek = noop\_llseek,

};

块设备文件默认的文件操作结构实例定义如下（/fs/block\_dev.c）：

const struct file\_operations **def\_blk\_fops** = {

.open = **blkdev\_open**,

.release = blkdev\_close,

.llseek = block\_llseek,

.read\_iter = blkdev\_read\_iter,

.write\_iter = blkdev\_write\_iter,

.mmap = generic\_file\_mmap,

.fsync = blkdev\_fsync,

.unlocked\_ioctl = block\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = compat\_blkdev\_ioctl,

#endif

.splice\_read = generic\_file\_splice\_read,

.splice\_write = iter\_file\_splice\_write,

};

以字符设备文件为例，在打开文件的系统调用最后将调用**inode->i\_fop->open()**函数完成文件的底层打开操作。字符设备def\_chr\_fops实例中定义的打开函数为**chrdev\_open()**，函数内根据设备号搜索字符设备驱动数据库，找到对应的cdev实例，调用ops成员指向的file\_operations实例中的open()函数，并将ops指向实例赋予文件file实例f\_op成员，如下图所示，此后对字符设备文件的操作将由驱动程序中定义的file\_operations实例中的函数完成。



块设备文件的打开操作与此类似，在通用的打开函数blkdev\_open()中根据设备号查找块设备驱动数据结构gendisk实例，再调用底层块设备定义的打开操作函数激活设备，完成设备的打开操作。

块设备文件file实例f\_op成员指向默认的def\_blk\_fops实例，也就是说所有的块设备文件采用相同的文件操作结构实例，而不像字符设备一样由驱动程序定义文件操作结构实例。通过块设备文件访问的是裸块设备，而内核通常是通过文件系统间接地访问块设备。

后面两章讲解字符设备和块设备驱动程序时再详细介绍设备文件的操作。

## 8.3总线与设备类

通用驱动模型中总线由bus\_type结构体表示，也可以表示虚拟的总线。总线实例由内核（驱动程序）静态定义，总线bus\_type管理挂载在总线上的设备device和驱动device\_driver实例。

当向总线注册设备或驱动时，总线负责查找匹配的驱动或设备，匹配成功则调用驱动中的probe()函数，

创建并注册设备驱动数据结构实例（加载设备驱动程序）。

设备类用于管理同一类型的设备，同一类型设备可能挂接在不同的总线上。设备类定义了同类型设备的公共属性等。

本节介绍总线和设备类的定义（创建）和注册。

### 8.3.1总线

#### 1数据结构

总线bus\_type结构体定义在/include/linux/device.h头文件：

struct bus\_type {

const char **\*name**; /\*总线名称，导出到sysfs文件系统\*/

const char \*dev\_name; /\*默认设备名称\*/

struct device \***dev\_root**; /\*指向表示总线自身的device实例\*/

struct device\_attribute \*dev\_attrs; /\*总线设备的默认属性，device\_attribute数组\*/

const struct attribute\_group \*\*bus\_groups; /\*默认的总线属性，注册总线时添加\*/

const struct attribute\_group \*\*dev\_groups; /\*总线设备的默认属性（attribute指针数组）\*/

const struct attribute\_group \*\*drv\_groups; /\*总线驱动的默认属性（attribute指针数组）\*/

**int (\*match) (struct device \*dev, struct device\_driver \*drv);**

/\*检查指定device和device\_driver实例是否匹配，匹配成功返回非零值\*/

**int (\*uevent) (struct device \*dev, struct kobj\_uevent\_env \*env)**;

/\*添加、移除设备等时机调用，用于向uevent事件添加环境变量\*/

i**nt (\*probe) (struct device \*dev)**;

/\*设备驱动匹配成功时调用，用于注册设备驱动数据结构实例\*/

int (\*remove) (struct device \*dev); /\*设备从总线移除时调用\*/

void (\*shutdown) (struct device \*dev); /\*关机时调用，用于退出设备\*/

int (\*online)(struct device \*dev); /\*重新连接设备\*/

int (\*offline)(struct device \*dev); /\*热插拔移除设备\*/

int (\*suspend)(struct device \*dev, pm\_message\_t state); /\*设备进入睡眠状态时调用\*/

int (\*resume)(struct device \*dev); /\*唤醒设备时调用\*/

const struct dev\_pm\_ops **\*pm**; /\*总线功耗管理操作，调用驱动定义的dev\_pm\_ops操作\*/

const struct iommu\_ops \*iommu\_ops; /\*总线IOMMU特定操作，用于具有MMU的外设\*/

struct subsys\_private **\*p**; /\*私有数据结构，用于管理设备和驱动等\*/

struct lock\_class\_key lock\_key; /\*锁\*/

};

bus\_type结构体中主要成员简介如下：

●**dev\_root：**指向表示总线自身的device实例，在调用subsys\_system\_register()函数注册子系统时，在sysfs文件系统/sys/devices/system/目录下创建表示总线的device实例。

●**match()：**在向总线添加设备时，将扫描总线下的驱动，对每个驱动调用match()函数判断设备与扫描的驱动是否匹配，匹配成功match()返回非零值，否则返回0。同理，当向总线添加驱动时，扫描总线上的设备，对每个设备调用match()函数，匹配成功返回非零值，否则返回0。

●**probe()：**向总线添设备或驱动时，设备和驱动匹配成功则调用此函数，完成驱动程序的加载。如果总线没有定义probe()函数，设备和驱动匹配成功后将调用device\_driver实例中的probe()函数。

驱动程序加载主要是向设备驱动数据库注册cdev或gendisk实例。硬件设备的激活（初始化）在打开设备文件时，调用设备文件操作结构file\_operations实例中的open()函数完成。

●**pm：**设备功耗管理结构dev\_pm\_ops指针，dev\_pm\_ops结构体定义在/include/linux/pm.h头文件，总线dev\_pm\_ops实例内函数将回调驱动device\_driver实例中关联的dev\_pm\_ops实例中的函数。

●**bus\_groups：**指向attribute\_group结构体指针数组，表示总线的默认属性组，属性文件添加到总线目录项下。attribute\_group结构体定义如下（/include/linux/sysfs.h）：

struct attribute\_group {

const char \*name; /\*属性组名称\*/

umode\_t (\*is\_visible)(struct kobject \*,struct attribute \*, int); /\*判断属性是否可见\*/

struct attribute \*\*attrs; /\*指向属性指针数组\*/

struct bin\_attribute \*\*bin\_attrs; /\*指向二进制属性指针数组\*/

};

attribute\_group结构体中包含的是一组属性，若name不为NULL，则向sysfs添加属性时创建名称为name的目录项，属性文件添加到此目录项下。

int **sysfs\_create\_groups**(struct kobject \*kobj,const struct attribute\_group \*\*groups)：函数用于向kobj实例（目录项）下添加属性组，函数定义在/fs/sysfs/group.c文件内，源代码请读者自行阅读。

●**p：**subsys\_private结构体指针，subsys\_private用于表示总线bus\_type和设备类class的私有数据，结构体定义在/drivers/base/base.h头文件：

struct subsys\_private {

struct kset **subsys**; /\*内嵌kset实例，在sysfs内表示总线/设备类\*/

struct kset \***devices\_kset**; /\*指向设备集合kset实例，管理总线下设备device实例\*/

struct list\_head **interfaces**; /\*链接subsys\_interface或class\_interface实例\*/

struct mutex mutex; /\*互斥量，保护设备及interfaces链表\*/

struct kset \***drivers\_kset**; /\*指向驱动集合kset实例，跟踪总线上驱动device\_driver实例\*/

struct klist **klist\_devices**; /\*总线上设备device实例链表\*/

struct klist **klist\_drivers**; /\*总线上驱动device\_driver实例链表\*/

struct blocking\_notifier\_head bus\_notifier; /\*通知链\*/

unsigned int drivers\_autoprobe:1; /\*添加设备时是否自动探测驱动（匹配驱动），默认是\*/

struct bus\_type \*bus; /\*指向总线bus\_type实例\*/

struct kset glue\_dirs; /\*\*/

struct class \*class; /\*指设备类class实例\*/

};

subsys\_private结构体中subsys成员用于将总线/设备类实例导出到sysfs文件系统，并作为其下设备和驱动的父目录项。devices\_kset和drivers\_kset指向kset实例分别用于跟踪总线下的device和device\_driver实例，并在sysfs文件系统中做为设备和驱动实例的父目录。链表成员klist\_devices和klist\_drivers分别管理device和device\_driver实例，在匹配操作中便于扫描device和device\_driver实例。

#### 2注册总线

总线bus\_type实例必须由内核（总线驱动）代码静态定义，内核没有提供动态创建bus\_type实例的方法。总线驱动程序中创建bus\_type实例后需要向内核注册。在讲解总线注册前，先看一下总线初始化函数。

总线初始化函数buses\_init()在/drivers/base/bus.c文件内实现，代码如下：

int \_\_init buses\_init(void)

{

**bus\_kset** = kset\_create\_and\_add("bus", **&bus\_uevent\_ops**, NULL); /\*创建目录/sys/bus\*/

...

**system\_kset** = kset\_create\_and\_add("system", NULL, &**devices\_kset->kobj)**;

/\*创建目录/sys/devices/system\*/

...

return 0;

}

buses\_init()函数创建名称为"bus"的kset实例，导出到在/sys/目录下，即/sys/bus/为所有总线实例的父目录，用于跟踪注册的总线bus\_type实例。创建名称为"system"的kset实例，导出到在/sys/devices目录下，即创建/sys/devices/system目录。

"bus"名称的kset实例，其指向的kset\_uevent\_ops结构体实例bus\_uevent\_ops定义在/drivers/base/bus.c文件内：

static const struct kset\_uevent\_ops bus\_uevent\_ops = {

.filter = **bus\_uevent\_filter**,

};

bus\_uevent\_filter()函数定义如下：

static int bus\_uevent\_filter(struct kset \*kset, struct kobject \*kobj)

{

struct kobj\_type \*ktype = get\_ktype(kobj);

if (ktype == &**bus\_ktype**) /\*若为总线实例总返回1，表示发送uevent事件，否则返回0\*/

return 1;

return 0;

}

在向内核注册总线实例时，进行以下赋值：

bus\_type.p.**subsys.kobj.kset = bus\_kset**

bus\_type.p.**subsys.kobj.ktype = &bus\_ktype**

因此对于bus\_type.p.**subsys**实例bus\_uevent\_filter()函数总是返回1，表示内核不过滤总线的uevent事件，总是允许发送uevent事件。

注册总线**bus\_register()**函数定义在/drivers/base/bus.c文件内，注册成功返回0，否则返回错误码：

int bus\_register(struct bus\_type \*bus)

{

int retval;

struct subsys\_private \*priv;

struct lock\_class\_key \*key = &bus->lock\_key;

priv = **kzalloc(sizeof(struct subsys\_private), GFP\_KERNEL)**; /\*创建subsys\_private实例\*/

if (!priv)

return -ENOMEM;

**priv->bus = bus;**  /\*指向总线实例\*/

**bus->p = priv;**  /\*指向subsys\_private实例\*/

BLOCKING\_INIT\_NOTIFIER\_HEAD(&priv->bus\_notifier); /\*初始化通知链\*/

**retval = kobject\_set\_name(&priv->subsys.kobj, "%s", bus->name)**; /\*设置总线kobject名称\*/

if (retval)

goto out;

**priv->subsys.kobj.kset = bus\_kset;** /\*父节点bus\_set在初始化时创建，总线位于/sys/bus/目录下\*/

**priv->subsys.kobj.ktype = &bus\_ktype;** /\*设置总线kobject类型为bus\_type\*/

**priv->drivers\_autoprobe = 1**; /\*允许自动探测驱动\*/

**retval = kset\_register(&priv->subsys);** /\*注册总线kset实例，导出路径为/sys/bus/*bus->name*\*/

if (retval)

goto out;

**retval = bus\_create\_file(bus, &bus\_attr\_uevent);**  /\*添加uevent总线属性，总线属性后面介绍\*/

... /\*错误处理\*/

**priv->devices\_kset = kset\_create\_and\_add("devices", NULL, &priv->subsys.kobj);**

/\*创建总线设备“devices”子目录，/sys/bus/*bus->name*/devices/\*/

... /\*错误处理\*/

**priv->drivers\_kset = kset\_create\_and\_add("drivers", NULL, &priv->subsys.kobj);**

/\*创建总线驱动“drivers”子目录，/sys/bus/*bus->name*/drivers/\*/

... /\*错误处理\*/

**INIT\_LIST\_HEAD(&priv->interfaces)**; /\*初始化子系统接口链表头\*/

\_\_mutex\_init(&priv->mutex, "subsys mutex", key);

**klist\_init(&priv->klist\_devices, klist\_devices\_get, klist\_devices\_put);**  /\*初始化device链表\*/

/\*函数参数用于增加/减小device实例引用计数值\*/

**klist\_init(&priv->klist\_drivers, NULL, NULL)**; /\*初始化device\_driver链表\*/

**retval = add\_probe\_files(bus);** /\*添加总线drivers\_probe和drivers\_autoprobe属性\*/

if (retval)

goto bus\_probe\_files\_fail;

**retval = bus\_add\_groups(bus, bus->bus\_groups);** /\*添加总线默认属性，bus->bus\_groups\*/

...

return 0;

...

}

注册总线bus\_register()函数主要完成的工作如下：

1. 创建subsys\_private结构体实例，初始化，将subsys\_private.subsys.kobj导出到/sys/bus/目录下。
2. 在subsys\_private.subsys下分别创建跟踪设备和驱动实例的kset，导出/sys/bus/*bus->name*/devices和/sys/bus/*bus->name*/drivers。
3. 向总线**subsys\_private.subsys.kobj**添加uevent，drivers\_probe，drivers\_autoprobe属性以及总线实例定义的默认属性组bus->bus\_groups中定义的属性。

下图示意了注册总线在sysfs中导出的目录项和文件：



内核在/drivers/base/bus.c文件内定义了注册子系统的函数int **subsys\_system\_register**(struct bus\_type \*subsys,const struct attribute\_group \*\*groups)，参数subsys表示总线实例指针，groups指向总线属性指针数组。此函数内完成subsys总线的注册，创建表示总线的device实例（将总线视为设备），导出到sysfs文件系统/sys/devices/system/目录下，device实例指针赋予subsys->dev\_root成员，函数源代码请读者自行阅读。

void bus\_unregister(struct bus\_type \*bus)函数用于注销总线。

#### 3总线属性

在注册总线时，内核会在跟踪总线的subsys\_private.subsys.kobj实例下添加uevent、drivers\_autoprobe、和drivers\_probe属性以及总线默认属性组bus->bus\_groups中定义的属性。属性在sysfs（kernfs）文件系统中表现为文件，用户可对属性文件进行读写，从而获取总线信息或对总线进行控制。

总线属性定义如下（/include/linux/device.h）：

struct **bus\_attribute** {

**struct attribute attr**; /\*通用属性结构体成员\*/

ssize\_t (\*show) (struct bus\_type \*bus, char \*buf); /\*总线属性读函数\*/

ssize\_t (\*store) (struct bus\_type \*bus, const char \*buf, size\_t count); /\*总线属性写函数\*/

};

内核在/include/linux/device.h头文件内实现了定义总线属性的宏，例如：

#define BUS\_ATTR(\_name, \_mode, \_show, \_store) \ /\*定义读写属性，\_show、\_store为读写函数\*/

struct bus\_attribute bus\_attr\_##\_name = \_\_ATTR(\_name, \_mode, \_show, \_store)

/\*\_\_ATTR()宏定义在/include/linux/sysfs.h头文件\*/

#define BUS\_ATTR\_RW(\_name) \ /\*定义读写属性，只需指定属性名称，只有文件属主在写权限\*/

struct bus\_attribute bus\_attr\_##\_name = \_\_ATTR\_RW(\_name)

/\*读写函数名为\_name##\_show(), \_name##\_store()\*/

#define BUS\_ATTR\_RO(\_name) \ /\*定义只读属性，读函数名为\_name##\_store()\*/

struct bus\_attribute bus\_attr\_##\_name = \_\_ATTR\_RO(\_name)

BUS\_ATTR()等宏只是定义bus\_attribute实例，其读写函数需要另外定义（后面设备类、设备、驱动的属性也是这样）。

在注册总线函数中跟踪总线的kobect类型定义为subsys.kobj.ktype = **&bus\_ktype**，bus\_ktype实例定义在/drivers/base/bus.c文件内：

static struct kobj\_type bus\_ktype = {

.sysfs\_ops = &**bus\_sysfs\_ops**, /\*读写属性操作结构\*/

.release = bus\_release,

};

static const struct sysfs\_ops **bus\_sysfs\_ops** = {

.show = **bus\_attr\_show**, /\*读属性函数\*/

.store = **bus\_attr\_store**, /\*写属性函数\*/

};

用户读写总线属性文件时，将先调用bus\_sysfs\_ops实例中的show()和store()函数，再调用具体总线属性bus\_attribute实例中定义的show()和store()函数，如下图所示。



bus\_sysfs\_ops实例中的show()和store()函数定义如下：

static ssize\_t bus\_attr\_show(struct kobject \*kobj, struct attribute \*attr,char \*buf)

{

struct bus\_attribute \*bus\_attr = to\_bus\_attr(attr); /\*attribute指针转bus\_attribute指针\*/

struct subsys\_private \*subsys\_priv = to\_subsys\_private(kobj);

/\*容器机制，获取subsys\_private实例指针\*/

ssize\_t ret = 0;

if (bus\_attr->show)

ret = **bus\_attr->show(subsys\_priv->bus, buf);** /\*调用具体总线属性读函数\*/

return ret;

}

static ssize\_t bus\_attr\_store(struct kobject \*kobj, struct attribute \*attr,const char \*buf, size\_t count)

{

struct bus\_attribute \*bus\_attr = to\_bus\_attr(attr);

struct subsys\_private \*subsys\_priv = **to\_subsys\_private(kobj)**;

ssize\_t ret = 0;

if (bus\_attr->store)

ret = **bus\_attr->store(subsys\_priv->bus, buf, count);** /\*调用具体总线属性写函数\*/

return ret;

}

读和写属性函数通过容器机制，由属性attribute实例获取总线属性bus\_attribute实例指针，由kobject实例指针获取subsys\_private实例指针（进而获取总线实例指针），然后调用bus\_attribute实例中定义的具体属性读写函数完成属性的读写操作。

添加总线属性的add\_probe\_files()和bus\_create\_file()函数等，最终都是调用sysfs\_create\_file()函数，在总线目录项下添加属性文件。

在注册总线时默认添加uevent、drivers\_probe、drivers\_autoprobe总线属性以及总线定义的默认属性。

●uevent：只写属性，用于发送uevent事件。

●drivers\_probe：只写属性，用于为指定名称的设备探测驱动。

●drivers\_autoprobe：可读写属性，用于读写bus->p->drivers\_autoprobe成员值，写入非零值表示设置成员值，否则清零。drivers\_autoprobe成员值用于控制向总线添加设备时是否自动探测驱动。

#### 4子系统接口

子系统接口通常用于对总线下的设备执行某项特殊操作。子系统接口由subsys\_interface结构体表示，定义如下（/include/linux/device.h）：

struct subsys\_interface {

const char \*name; /\*名称\*/

struct bus\_type \*subsys; /\*指向总线\*/

struct list\_head node; /\*双链表节点，将实例添加到总线subsys->p->interfaces双链表\*/

int (\***add\_dev**)(struct device \*dev, struct subsys\_interface \*sif); /\*添加设备到接口\*/

int (\***remove\_dev**)(struct device \*dev, struct subsys\_interface \*sif);

};

注册子系统接口的subsys\_interface\_register()函数定义如下（/drivers/base/bus.c）：

int subsys\_interface\_register(struct subsys\_interface \*sif)

{

struct bus\_type \*subsys;

struct subsys\_dev\_iter iter; /\*/include/linux/device.h\*/

struct device \*dev;

...

subsys = **bus\_get(sif->subsys)**; /\*总线\*/

...

mutex\_lock(&subsys->p->mutex);

**list\_add\_tail(&sif->node, &subsys->p->interfaces)**; /\*添加到subsys\_private中双链表\*/

if (sif->add\_dev) {

subsys\_dev\_iter\_init(&iter, subsys, NULL, NULL); /\*遍历总线下设备\*/

while ((dev = subsys\_dev\_iter\_next(&iter)))

**sif->add\_dev(dev, sif)**; /\*添加设备到接口\*/

subsys\_dev\_iter\_exit(&iter);

}

mutex\_unlock(&subsys->p->mutex);

return 0;

}

注册子系统接口时将对总线下所有设备调用sif->add\_dev(dev, sif)函数。

注销子系统接口的函数为subsys\_interface\_unregister()，将对总线下所有设备调用sif->remove\_dev(dev, sif)函数，请读者自行阅读源代码。

### 8.3.2设备类

设备类是对设备的高层次抽象，由class结构体表示，用于按功能对设备进行分类，一个设备类管理同一类型的设备，例如：输入设备类，块设备类等。

#### 1数据结构

设备类class结构体定义在/include/linux/device.h头文件：

struct class {

const char \*name; /\*设备类名称\*/

struct module \*owner; /\*模块指针\*/

struct class\_attribute \*class\_attrs; /\*设备类默认属性\*/

const struct attribute\_group \*\*dev\_groups; /\*设备类中设备的默认属性（组）\*/

struct kobject **\*dev\_kobj**; /\*默认指向/sys/dev/char/对应的kobject实例\*/

int (\***dev\_uevent**) (struct device \*dev, struct kobj\_uevent\_env \*env);

/\*添加、移除设备时，用于向uevent事件添加环境变量\*/

char \*(\***devnode**) (struct device \*dev, umode\_t \*mode);

/\*用于自动创建设备文件时，确定文件名称\*/

void (\***class\_release**) (struct class \*class); /\*释放设备类时调用此函数\*/

void (\*dev\_release) (struct device \*dev); /\*释放设备时调用此函数\*/

int (\*suspend) (struct device \*dev, pm\_message\_t state);

/\*用于使设备睡眠，一般为使设备进入低功耗状态\*/

int (\*resume) (struct device \*dev); /\*用于从睡眠模式唤醒设备\*/

const struct kobj\_ns\_type\_operations \*ns\_type; /\*sysfs用于确定命名空间\*/

const void \*(\*namespace) (struct device \*dev); /\*返回命名空间标签\*/

**const struct dev\_pm\_ops \*pm**; /\*设备类中设备默认功耗管理操作结构\*/

**struct subsys\_private \*p;** /\*设备类私有数据结构，与bus中私有数据结构相同\*/

};

#### 2创建设备类

内核提供了动态创建设备类的接口函数，函数内负责创建class实例并向内核注册。在介绍创建设备类函数之前，先简单看一下设备类的初始函数classes\_init()，函数在/drivers/base/class.c文件内定义：

int \_\_init classes\_init(void)

{

**class\_kset** = kset\_create\_and\_add("class", NULL, NULL); /\*/sys/class\*/

...

return 0;

}

初始化函数很简单，主要是创建名称为class的kset结构体实例，并导出到sysfs文件系统根目录下，即创建/sys/class/目录，表示设备类的集合。

创建设备类的接口函数class\_create()声明在/include/linux/device.h头文件：

#define **class\_create**(owner, name) \

({ \

static struct lock\_class\_key \_\_key; \

**\_\_class\_create(owner, name, &\_\_key)**; \

})

参数owner表示模块指针，name表示设备类名称。

class\_create()函数内调用的\_\_class\_create()函数在/drivers/base/class.c内实现，代码如下：

struct class \*\_\_class\_create(struct module \*owner, const char \*name,struct lock\_class\_key \*key)

{

struct class \*cls;

int retval;

**cls = kzalloc(sizeof(\*cls), GFP\_KERNEL)**; /\*创建class结构体实例\*/

... /\*错误处理\*/

**cls->name = name;**  /\*设备类名称\*/

**cls->owner = owner;**

**cls->class\_release = class\_create\_release;** /\*默认的释放设备类函数\*/

retval = **\_\_class\_register(cls, key);** /\*注册设备类\*/

...

return cls;

...

}

注册设备类的\_\_class\_register()函数定义如下：

int \_\_class\_register(struct class \*cls, struct lock\_class\_key \*key)

{

struct subsys\_private \*cp;

int error;

pr\_debug("device class '%s': registering\n", cls->name);

cp = **kzalloc(sizeof(\*cp), GFP\_KERNEL)**; /\*创建subsys\_private实例\*/

if (!cp)

return -ENOMEM;

**klist\_init(&cp->klist\_devices, klist\_class\_dev\_get, klist\_class\_dev\_put)**; /\*初始化设备链表\*/

INIT\_LIST\_HEAD(&cp->interfaces); /\*初始化链表\*/

**kset\_init(&cp->glue\_dirs);**  /\*初始化链表成员\*/

\_\_mutex\_init(&cp->mutex, "subsys mutex", key);

**error = kobject\_set\_name(&cp->subsys.kobj, "%s", cls->name);**

/\*设备类名称设为subsys.kobj名称\*/

... /\*错误处理\*/

if (!cls->dev\_kobj) /\*dev\_kobj为空则指向字符设备kobject，/sys/dev/char/\*/

**cls->dev\_kobj = sysfs\_dev\_char\_kobj;**  /\*/sys/dev/char/\*/

#if defined(CONFIG\_BLOCK) /\*如果选择了支持块设备配置选择项\*/

if (!sysfs\_deprecated || cls != &block\_class)

**cp->subsys.kobj.kset = class\_kset**; /\*不是block\_class则上级目录为/sys/class/\*/

#else

cp->subsys.kobj.kset = class\_kset; /\*没有配置支持块设备时，上级目录为/sys/class/\*/

#endif

**cp->subsys.kobj.ktype = &class\_ktype**; /\*设备类kobject类型\*/

**cp->class = cls;** /\*指向设备类\*/

**cls->p = cp;**  /\*指向subsys\_private实例\*/

**error = kset\_register(&cp->subsys)**; /\*注册kset，导出目录为/sys/class/*class\_name*\*/

... /\*错误处理\*/

**error = add\_class\_attrs(class\_get(cls))**;

/\*添加设备类默认属性，在/sys/class/*class\_nam*/下创建属性文件\*/

class\_put(cls);

return error;

}

设备类的注册比总线的注册简单许多，函数内创建设备类私有数据结构subsys\_private实例，并初始化，将跟踪class实例的class.subsys.kobj实例添加到class\_kset集合下（块设备类外），设备类在sysfs文件系统中导出目录/sys/class/*class\_name*/，函数最后创建设备类的默认属性文件，如下图所示。



设备类class实例也可以静态创建，随后调用**class\_register(class)**函数向内核注册，此函数内直接调用\_\_class\_register(cls, key)函数完成设备类实例的注册。

void class\_unregister(struct class \*cls)函数用于注销设备类。

#### 3设备类属性

设备类属性由class\_attribute结构表示，定义如下（/include/linux/device.h）：

struct class\_attribute {

struct attribute attr; /\*通用属性结构体\*/

ssize\_t (\*show) (struct class \*class, struct class\_attribute \*attr,char \*buf); /\*具体属性读函数\*/

ssize\_t (\*store) (struct class \*class, struct class\_attribute \*attr,const char \*buf, size\_t count);

/\*具体属性写函数\*/

};

内核在/include/linux/device.h头文件中实现了定义设备类属性的宏，例如：

#define CLASS\_ATTR(\_name, \_mode, \_show, \_store) \ /\*指定读写函数的读写属性\*/

struct class\_attribute **class\_attr\_##\_name** = \_\_ATTR(\_name, \_mode, \_show, \_store)

#define CLASS\_ATTR\_RW(\_name) \ /\*默认函数名称的读写属性\*/

struct class\_attribute class\_attr\_##\_name = \_\_ATTR\_RW(\_name)

#define CLASS\_ATTR\_RO(\_name) \ /\*默认函数名称的只读属性\*/

struct class\_attribute class\_attr\_##\_name = \_\_ATTR\_RO(\_name)

内核在注册设备类class实例时，其kobject实例类型为**subsys.kobj.ktype = &class\_ktype**，class\_ktype实例定义如下：

static struct kobj\_type **class\_ktype** = {

.sysfs\_ops= &**class\_sysfs\_ops**,

.release = class\_release,

.child\_ns\_type = class\_child\_ns\_type,

};

static const struct sysfs\_ops **class\_sysfs\_ops** = {

**.show = class\_attr\_show,**  /\*设备类属性读函数\*/

**.store = class\_attr\_store**, /\*设备类属性写函数\*/

};

class\_sysfs\_ops实例中定义的show()和store()函数将调用设备类属性class\_attribute中的show()和store()函数，如下图所示：



设备类属性读写函数与总线属性的读写函数类似，函数定义在/drivers/base/class.c文件内，源代码请读者自行阅读。

向设备类添加属性的函数最终调用sysfs\_create\_file\_ns()函数，在设备类对应的目录项下创建属性文件。

#### 4设备类接口

设备类与总线类似，也支持设备类接口。设备类接口由class\_interface结构体表示，定义如下：

struct class\_interface { /\*/include/linux/device.h\*/

struct list\_head node; /\*添加到设备类subsys->p->interfaces双链表\*/

struct class \*class; /\*指向设备类\*/

int (\*add\_dev) (struct device \*, struct class\_interface \*); /\*向接口添加设备\*/

void (\*remove\_dev) (struct device \*, struct class\_interface \*); /\*从接口移除设备\*/

};

注册、注销设备类接口的函数定义在/drivers/base/class.c文件内，原型如下，源代码请读者自行阅读：

int \_\_must\_check **class\_interface\_register**(struct class\_interface \*)：注册设备类接口，成功返回0。

void **class\_interface\_unregister**(struct class\_interface \*)：注销设备类接口。

## 8.4设备与驱动

在通用驱动模型中设备由device结构体表示，用于向驱动传递设备硬件信息，驱动由device\_driver结构体表示，用于驱动程序和设备的管理。

设备device和驱动device\_driver必须挂接在总线上，当向总线注册设备/驱动时，将扫描总线中的驱动/设备链表，寻找匹配的项。若设备与驱动匹配成功，则调用device\_driver实例中定义的探测函数probe()，注册设备驱动程序，即创建设备驱动数据结构实例并注册。

### 8.4.1创建管理目录

在通用驱动模型初始化函数中，将调用devices\_init()函数在sysfs中创建管理设备和驱动的目录，函数在/drivers/base/core.c文件内实现，代码简列如下：

int \_\_init devices\_init(void)

{

**devices\_kset** = kset\_create\_and\_add("devices", &**device\_uevent\_ops**, NULL); /\*创建/sys/devices/目录\*/

...

**dev\_kobj** = kobject\_create\_and\_add("dev", NULL); /\*创建/sys/dev/目录\*/

...

**sysfs\_dev\_block\_kobj** = kobject\_create\_and\_add("block", dev\_kobj); /\*创建/sys/dev/block/目录\*/

...

**sysfs\_dev\_char\_kobj** = kobject\_create\_and\_add("char", dev\_kobj); /\*创建/sys/dev/char/目录\*/

...

return 0;

...

}

devices\_init()函数比较简单主要是在sysfs文件系统中创建kset和kobject实例。

这里需要注意的是devices\_kset实例的创建，在向内核添加device实例时，默认将device实例添加到**devices\_kset**集合中，总线和设备类下的device是到devices\_kset下device实例的符号链接。

devices\_init()函数在sysfs中创建的目录项如下图所示：



devices\_init()函数中创建的devices\_kset实例，其uevent\_ops成员被赋予device\_uevent\_ops，此实例定义在/drivers/base/core.c文件内：

static const struct kset\_uevent\_ops device\_uevent\_ops = {

.filter = dev\_uevent\_filter, /\*对于device实例，返回1，不屏蔽uevent事件\*/

.name = dev\_uevent\_name, /\*获取产生uevent事件的总线或设备类的名称\*/

**.uevent = dev\_uevent,**  /\*向uevent事件添加环境变量的函数\*/

};

在添加/删除设备device实例时，会调用kobject\_uevent()函数发送uevent事件。由前面第7章介绍的kobject\_uevent()函数可知，将调用device实例所在集合kset实例关联kset\_uevent\_ops实例中的uevent()函数，即devices\_kset->uevent\_ops->uevent()函数，向uevent事件添加环境变量，也就是调用dev\_uevent()函数。

dev\_uevent()函数定义如下（/drivers/base/core.c）：

static int **dev\_uevent**(struct kset \*kset, struct kobject \*kobj, struct kobj\_uevent\_env \*env)

{

struct device \*dev = kobj\_to\_dev(kobj);

int retval = 0;

**if (MAJOR(dev->devt)) {**  /\*如果device实例devt成员中赋值了主设备号\*/

const char \*tmp;

const char \*name;

umode\_t mode = 0;

kuid\_t uid = GLOBAL\_ROOT\_UID;

kgid\_t gid = GLOBAL\_ROOT\_GID;

add\_uevent\_var(env, "MAJOR=%u", MAJOR(dev->devt)); /\*向uevent事件添加环境变量\*/

add\_uevent\_var(env, "MINOR=%u", MINOR(dev->devt)); /\*传递主从设备号信息\*/

name = **device\_get\_devnode(dev, &mode, &uid, &gid, &tmp)**; /\*设备名称\*/

if (name) {

add\_uevent\_var(env, "DEVNAME=%s", name);

if (mode)

add\_uevent\_var(env, "DEVMODE=%#o", mode & 0777);

if (!uid\_eq(uid, GLOBAL\_ROOT\_UID))

add\_uevent\_var(env, "DEVUID=%u", from\_kuid(&init\_user\_ns, uid));

if (!gid\_eq(gid, GLOBAL\_ROOT\_GID))

add\_uevent\_var(env, "DEVGID=%u", from\_kgid(&init\_user\_ns, gid));

kfree(tmp);

}

}

if (dev->type && dev->type->name) /\*设备类型名称\*/

add\_uevent\_var(env, "DEVTYPE=%s", dev->type->name);

**if (dev->driver)**  /\*关联了驱动\*/

add\_uevent\_var(env, "DRIVER=%s", dev->driver->name); /\*驱动名称\*/

**of\_device\_uevent(dev, env)**; /\*设备树相关\*/

if (dev->bus && dev->bus->uevent) {

**retval = dev->bus->uevent(dev, env)**; /\*调用总线uevent事件函数，添加环境变量\*/

...

}

if (dev->class && dev->class->dev\_uevent) {

**retval = dev->class->dev\_uevent(dev, env);** /\*调用设备类uevent事件函数，添加环境变量\*/

...

}

if (dev->type && dev->type->uevent) { /\*设备类型定义的uevent函数，添加环境变量\*/

**retval = dev->type->uevent(dev, env);**

...

}

return retval;

}

dev\_uevent()函数主要用于向uevent事件添加环境变量，包括设备所在总线、设备类、设备类型等需要传递的环境变量，各环境变量通过uevent机制传递到用户空间，用户空间进程（udev/mdev）获取环境变量后，可执行相应的操作，如加载模块、创建设备文件等。

### 8.4.2设备

在通用驱动模型中设备由device结构体表示，通常由板级（平台）相关代码定义并注册设备device实例。

#### 1数据结构

设备device结构体定义在/include/linux/device.h头文件内：

struct device {

struct device  **\*parent;** /\*指向父设备，通常是表示总线或主机控制器的device实例\*/

struct device\_private  **\*p;** /\*设备私有数据结构指针\*/

struct kobject **kobj;** /\*跟踪device实例的kobject实例，导出到sysfs文件系统\*/

const char \***init\_name**; /\*设备初始名称，可用于设备文件名称\*/

const struct device\_type  **\*type**; /\*设备类型，包含设备类型特定信息\*/

struct mutex mutex; /\*互斥量\*/

struct bus\_type **\*bus**; /\*所在总线实例指针\*/

struct device\_driver **\*driver;**  /\*匹配驱动实例指针\*/

void \***platform\_data**; /\*平台相关硬件信息，在板级文件中定义，如IO地址等\*/

void \***driver\_data**; /\*驱动程序中使用的数据结构\*/

struct dev\_pm\_info  **power**; /\*电源管理结构，/include/linux/pm.h\*/

struct dev\_power\_domain **\*pm\_domain**; /\*提供关机、停机等时机的回调函数\*/

#ifdef CONFIG\_PINCTRL

struct dev\_pin\_info \*pins; /\*引脚信息\*/

#endif

#ifdef CONFIG\_NUMA

...

#endif

u64 \*dma\_mask; /\* dma mask\*/

u64 coherent\_dma\_mask; / \* \*/

unsigned long dma\_pfn\_offset; /\*DMA内存偏移量相对于RAM\*/

struct device\_dma\_parameters \*dma\_parms;

struct list\_head dma\_pools; /\* dma pools (if dma'ble) \*/

struct dma\_coherent\_mem \*dma\_mem; /\* internal for coherent memoverride \*/

#ifdef CONFIG\_DMA\_CMA

...

#endif

struct dev\_archdata archdata;

/\*体系结构相关数据结构实例，/arch/mips/include/asm/device.h\*/

struct device\_node \***of\_node**; /\*对应的设备树节点\*/

struct fwnode\_handle \*fwnode; /\*固件相关设备节点\*/

dev\_t **devt;**  /\*设备号\*/

u32 **id;** /\*设备在总线中的编号\*/

spinlock\_t devres\_lock; /\*自旋锁用于保护设备资源\*/

struct list\_head  **devres\_head;** /\*设备资源devres实例双链表\*/

struct klist\_node **knode\_class;** /\*用于将设备链入设备类中链表\*/

struct class **\*class**; /\*设备所属设备类指针\*/

const struct attribute\_group **\*\*groups;** /\*设备属性指针数组 \*/

void (\*release) (struct device \*dev); /\*释放设备时调用\*/

struct iommu\_group \*iommu\_group; /\*IOMMU group\*/

bool offline\_disabled:1; /\*置1表示设备持久在线\*/

bool offline:1; /\*设备下线时置1\*/

};

device结构体主要成员简介如下：

**●parent：**指向父设备，通常是表示总线或主机控制器的device实例。内核中将总线、设备控制器等也视为设备注册到内核，device实例在内核中组成层次的父子结构。

**●kobj：**跟踪device实例的kobject结构体成员，导出到sysfs文件系统。

**●driver：**指向匹配驱动device\_driver实例。

**●platform\_data：**指向表示设备硬件信息的数据结构，如IO地址、中断号等，结构体在板级文件中定义，传递给驱动程序。

**●driver\_data：**指向驱动程序中数据结构。

**●devt：**设备号，含主从设备号。

●**type：**设备类型device\_type结构体指针。device\_type结构体定义在/include/linux/device.h头文件：

struct device\_type {

const char \*name; /\*设备类型名称\*/

const struct attribute\_group \*\*groups; /\*设备属性指针数组\*/

int (\*uevent) (struct device \*dev, struct kobj\_uevent\_env \*env); /\*向uevent事件添加环境变量函数\*/

char \*(\*devnode) (struct device \*dev, umode\_t \*mode,kuid\_t \*uid, kgid\_t \*gid);

/\*获取设备文件名称的函数\*/

void (\*release) (struct device \*dev); /\*释放设备函数\*/

const struct dev\_pm\_ops \*pm; /\*电源管理相关数据结构\*/

};

device\_type结构体表示的设备类型不是前面介绍的设备类，它是比设备类更细的划分，如鼠标、分区等。

●**p**：device私有数据结构device\_private指针，device\_private结构体定义在/drivers/base/base.h头文件，表示设备私有信息：

struct device\_private {

struct klist **klist\_children**; /\*链接子设备的链表头\*/

struct klist\_node **knode\_parent**; /\*链接同一父设备下兄弟设备\*/

struct klist\_node **knode\_driver**; /\*链接到驱动device\_driver中链表\*/

struct klist\_node **knode\_bus**; /\*链接到总线下的设备链表\*/

struct list\_headdeferred\_probe; /\*用于重试绑定驱动\*/

struct device **\*device;** /\*指向表示设备device实例\*/

};

由上可知device实例存在于以下管理结构中（如下图所示）：

（1）通过device\_private.knode\_parent成员添加到父设备device\_private.klist\_children链表。

（2）通过device\_private.knode\_bus添加到总线的设备链表。

（3）通过knode\_class成员添加到设备类的设备链表。

（4）通过device\_private.knode\_driver添加到匹配驱动的设备链表。



#### 2注册设备

device实例通常在板级（平台）相关代码中定义，或根据设备树节点（见下节）创建，然后注册到内核。

**device\_register(struct device \*dev)**函数用于向内核注册设备。注册设备就是将device实例添加到各种管理结构中，导出到sysfs文件系统，并探测总线上匹配的驱动，匹配成功则调用总线或驱动的probe()函数，完成设备驱动程序的注册。

device\_register(struct device \*dev)函数定义在/drivers/base/core.c文件内，代码如下：

int device\_register(struct device \*dev)

{

**device\_initialize(dev)**; /\*初始化设备，/drivers/base/core.c\*/

return **device\_add(dev)**; /\*添加设备，/drivers/base/core.c\*/

}

注册设备device实例主要分两步，一是初始化设备device实例，二是将设备device实例添加到各种管理结构中，导出到sysfs文件系统，并探测驱动等。

注销device实例的函数为void **device\_unregister**(struct device \*dev)。

##### ■初始化device

初始化device实例device\_initialize(dev)函数定义在/drivers/base/core.c文件内，代码如下：

void device\_initialize(struct device \*dev)

{

**dev->kobj.kset = devices\_kset;** /\*指定集合为devices\_kset（父节点）\*/

kobject\_init(&dev->kobj, &**device\_ktype**); /\*kobj\_type类型指定为device\_ktype\*/

INIT\_LIST\_HEAD(&dev->dma\_pools);

mutex\_init(&dev->mutex);

lockdep\_set\_novalidate\_class(&dev->mutex);

spin\_lock\_init(&dev->devres\_lock);

INIT\_LIST\_HEAD(&dev->devres\_head);

device\_pm\_init(dev); /\*电源管理结构初始化，/drivers/base/power/power.h\*/

set\_dev\_node(dev, -1); /\*设置内存节点，没有选择NUMA选项为空操作\*/

}

初始化函数主要完成device实例各成员的初始化，需要注意的是跟踪实例的kobject结构体成员，其kset成员指向devices\_kset，kobj\_type类型为device\_ktype。

device\_initialize()初始化的device实例在没有指定父设备的情况下将导出到/sys/devices/目录下，指定了父设备则导出到父设备对应的目录项下。

###### ●设备属性

device实例导出到sysfs为目录项，其下属性表现为文件。属性文件的读写操作由kobj\_type实例实现，device.kobj关联的kobj\_type实例为device\_ktype。

device\_ktype实例定义在/drivers/base/core.c文件内：

static struct kobj\_type **device\_ktype** = {

.release = device\_release,

.sysfs\_ops = &**dev\_sysfs\_ops,**  /\*读写设备属性操作结构\*/

.namespace = device\_namespace,

};

读写设备属性的通用操作dev\_sysfs\_ops实例定义如下（/drivers/base/core.c）：

static const struct sysfs\_ops dev\_sysfs\_ops = {

**.show = dev\_attr\_show**, /\*调用具体设备属性定义的读函数\*/

**.store = dev\_attr\_store**, /\*调用具体设备属性定义的写函数\*/

};

设备属性由device\_attribute结构体表示，定义如下（/include/linux/device.h）：

struct device\_attribute {

struct attribute attr; /\*通用属性结构体\*/

ssize\_t (\*show) (struct device \*dev, struct device\_attribute \*attr,char \*buf);

ssize\_t (\*store) (struct device \*dev, struct device\_attribute \*attr, const char \*buf, size\_t count);

};

在/include/linux/device.h头文件内实现了定义设备属性的宏，例如：

#define DEVICE\_ATTR(\_name, \_mode, \_show, \_store) \ /\*指定读写函数的设备属性\*/

struct device\_attribute dev\_attr\_##\_name = \_\_ATTR(\_name, \_mode, \_show, \_store)

#define DEVICE\_ATTR\_RW(\_name) \ /\*采用默认读写函数名称的设备属性\*/

struct device\_attribute dev\_attr\_##\_name = \_\_ATTR\_RW(\_name)

dev\_sysfs\_ops实例中的设备属性读写函数与总线、设备类属性读写函数相似，也是调用具体设备属性中定义的读写函数完成操作，如下图所示。



注意上图中标注的dev\_uevent()函数，在对device实例进行添加、删除等操作时，触发的uevent事件将调用dev\_uevent()函数向uevent事件添加环境变量，函数内又将调用设备所在总线、所属设备类和设备类型等实例中的uevent()函数向uevent事件添加环境变量，详见本节开头。

##### ■添加device

注册device实例的第二步是调用**device\_add(dev)**函数向内核添加device实例，device\_add(dev)函数是一个非常重要的函数，不光是在device\_register(dev)函数中会被调用。

**device\_add(dev)**函数完成的主要工作如下：

（1）为设备分配并初始化device\_private实例。

（2）确定设备名称，可能用于设备文件命名。

（3）确定父设备，并向sysfs添加device.kobj实例。如果存在父设备，则device.kobj导出到父设备在sysfs中的目录项下，否则导出到/sys/devices/目录下。假设设备导出目录项名称为dev\_name。

（4）在设备目录项下创建到设备类的符号链接subsystem（可能被后面的总线覆盖），创建到父设备的符号链接device（如果指定了父设备），在设备类下创建到设备的符号链接dev\_name。

（5）在设备目录项下创建到总线的符号链接subsystem， 在总线/sys/bus/*bus\_name*/devices/目录下创建到设备的符号链接dev\_name。

（6）在/sys/dev/char/或/sys/dev/block/目录下创建到设备的符号链接，名称为“主设备号：从设备号”。

（7）向设备目录项下添加属性文件，包括设备自带的默认属性，以及所在总线，所属设备类，设备类型赋予设备的默认属性。

（8）将device实例添加到各管理结构中，包括总线、设备类中的设备链表，父设备中的子设备链表等。

（9）自动创建设备文件，触发uevent事件。

（10）探测设备驱动，找到匹配驱动则加载设备驱动程序，并创建设备与驱动之间的符号链接，将device实例添加到驱动device\_driver实例中设备链表。

device\_add(dev)函数创建的目录项、符号链接、属性文件如下图所示：



下面来看一下device\_add(dev)函数的实现，函数定义在/drivers/base/core.c文件内：

int device\_add(struct device \*dev)

{

struct device \*parent = NULL;

struct kobject \*kobj;

struct class\_interface \*class\_intf;

int error = -EINVAL;

dev = get\_device(dev); /\*增加dev->kobj引用计数\*/

...

if (!dev->p) {

error = **device\_private\_init(dev)**; /\*分配并初始化**device\_private**实例\*/

...

}

if (**dev->init\_name**) { /\*设置了名称，可能用于设备文件名称\*/

**dev\_set\_name(dev, "%s", dev->init\_name)**; /\*设置init\_name至dev->kobj.name成员\*/

dev->init\_name = NULL;

}

if (!dev\_name(dev) && dev->bus && dev->bus->dev\_name) /\*如果没有设置设备名称\*/

dev\_set\_name(dev, "%s%u", dev->bus->dev\_name, dev->id);

/\*名称设为bus->dev\_name+dev->id\*/

... /\*错误处理\*/

parent = get\_device(dev->parent);

kobj = get\_device\_parent(dev, parent);

if (kobj) /\*如果存在父设备\*/

**dev->kobj.parent = kobj**; /\*指向父设备kobject\*/

if (parent)

set\_dev\_node(dev, dev\_to\_node(parent)); /\*没有选择NUMA选项为空操作\*/

error = **kobject\_add(&dev->kobj, dev->kobj.parent, NULL)**;

/\*向内核添加dev->kobj实例，创建设备对应的目录项，名称为init\_name\*/

...

/\* notify platform of device entry \*/

if (platform\_notify)

platform\_notify(dev);

error = **device\_create\_file(dev, &dev\_attr\_uevent)**;

/\*在设备目录项下添加uevent属性文件，写操作触发uevent事件\*/

...

error = **device\_add\_class\_symlinks(dev)**; /\*/drivers/base/core.c\*/

/\*在设备类创建下创建到设备的符号链接，创建到设备类、父设备的符号链接\*/

...

error = **device\_add\_attrs(dev)**; /\*/drivers/base/core.c\*/

/\*添加class->dev\_groups，device\_type->groups，device->groups指定的设备属性文件\*/

...

error = **bus\_add\_device(dev)**; /\*向总线添加设备，见下文，/drivers/base/bus.c\*/

...

error = dpm\_sysfs\_add(dev);

/\*配置了PM选项时注册设备电源管理属性，/drivers/base/power/sysfs.c\*/

...

device\_pm\_add(dev); /\*将设备添加到电源管理结构中，/drivers/base/power/main.c\*/

if (**MAJOR(dev->devt)**) { /\*如果设置了主设备号，则自动创建设备文件\*/

error = device\_create\_file(dev, &dev\_attr\_dev); /\*增加dev属性，读属性显示设备号\*/

...

error = **device\_create\_sys\_dev\_entry(dev)**; /\*/drivers/base/core.c\*/

/\*在/sys/dev/char或/sys/dev/block目录下创建到device实例的符号链接\*/

...

**devtmpfs\_create\_node(dev)**; /\*在/dev/（devtmpfs）目录下创建设备文件，见本章上文\*/

}

if (dev->bus) /\*执行总线中通知链\*/

blocking\_notifier\_call\_chain(&dev->bus->p->bus\_notifier,BUS\_NOTIFY\_ADD\_DEVICE, dev);

**kobject\_uevent(&dev->kobj, KOBJ\_ADD)**; /\*向用户空间发出uevent事件，传递环境变量\*/

**bus\_probe\_device(dev)**; /\*探测设备驱动，见下文，/drivers/base/bus.c\*/

if (parent)

klist\_add\_tail(&dev->p->knode\_parent,&parent->p->klist\_children); /\*添加到父设备的子链表\*/

if (dev->class) {

mutex\_lock(&dev->class->p->mutex);

**klist\_add\_tail(&dev->knode\_class,&dev->class->p->klist\_devices)**;

/\*将设备添加到设备类的设备链表\*/

list\_for\_each\_entry(class\_intf,&dev->class->p->interfaces, node)

if (class\_intf->add\_dev)

class\_intf->add\_dev(dev, class\_intf); /\*将设备添加到设备类接口\*/

mutex\_unlock(&dev->class->p->mutex);

}

done:

put\_device(dev);

return error;

...

}

device\_add(dev)函数的主要工作前面已经介绍过了。下面介绍一下向总线添加设备的bus\_add\_device()函数和探测设备驱动的bus\_probe\_device()函数的定义。

###### ●向总线添加设备

bus\_add\_device()函数用于将设备添加到总线，函数定义在/drivers/base/bus.c文件内，代码如下：

int bus\_add\_device(struct device \*dev)

{

struct bus\_type \*bus = bus\_get(dev->bus);

int error = 0;

if (bus) {

pr\_debug("bus: '%s': add device %s\n", bus->name, dev\_name(dev));

error = **device\_add\_attrs(bus, dev)**; /\*/drivers/base/bus.c\*/

/\*添加总线定义的设备默认属性，bus->dev\_attrs（device\_attribute数组）\*/

if (error)

goto out\_put;

error = **device\_add\_groups(dev, bus->dev\_groups)**;

/\*添加总线定义的设备属性，attribute指针数组，/drivers/base/core.c\*/

if (error)

goto out\_id;

error = **sysfs\_create\_link(&bus->p->devices\_kset->kobj,&dev->kobj, dev\_name(dev))**;

/\*创建/sys/bus/*bus\_name*/devices/*dev\_name*符号链接，到设备目录项\*/

if (error)

goto out\_groups;

error = **sysfs\_create\_link(&dev->kobj,&dev->bus->p->subsys.kobj, "subsystem")**;

/\*在设备目录项下创建到总线的符号链接，/sys/devices/*dev\_name*/subsystem\*/

if (error)

goto out\_subsys;

**klist\_add\_tail(&dev->p->knode\_bus, &bus->p->klist\_devices)**;

/\*将设备添加到总线bus->p->klist\_devices设备链表\*/

}

return 0;

...

}

bus\_add\_device()函数主要是向设备目录项下添加总线定义的设备属性，创建符号链接，将设备添加到总线中的设备链表等，这里还没有将设备添加到子系统接口（在bus\_probe\_device()函数中添加）。

###### ●匹配驱动

device\_add(dev)函数中调用**bus\_probe\_device()**函数完成添加设备最后一项非常重要的工作，那就是为设备探测驱动。

bus\_probe\_device()函数将扫描总线上的驱动device\_driver实例链表，逐个调用**bus->match(dev, drv)**函数判断驱动与设备是否匹配，若匹配则调用总线**bus->probe()**函数或驱动**device\_drive->probe()**函数完成设备驱动程序的加载（不再扫描后面的device\_driver实例），即注册设备驱动数据结构实例。

bus\_probe\_device()函数定义在/drivers/base/bus.c文件内，代码如下：

void bus\_probe\_device(struct device \*dev)

{

struct bus\_type \*bus = dev->bus;

struct subsys\_interface \*sif;

if (**!bus**)

return;

if (**bus->p->drivers\_autoprobe**) /\*如果总线允许自动探测驱动\*/

**device\_initial\_probe(dev)**; /\*匹配设备驱动，/drivers/base/dd.c\*/

mutex\_lock(&bus->p->mutex);

list\_for\_each\_entry(sif, &bus->p->interfaces, node)

if (sif->add\_dev)

sif->add\_dev(dev, sif); /\*将设备添加到子系统接口\*/

mutex\_unlock(&bus->p->mutex);

}

bus\_probe\_device()函数判断总线是否设置了允许自动探测驱动，如果是则调用device\_initial\_probe()函数完成驱动的探测，随后还将设备添加到总线中子系统接口。

device\_initial\_probe(struct device \*dev)函数定义在/drivers/base/dd.c文件内，代码如下：

void device\_initial\_probe(struct device \*dev)

{

**\_\_device\_attach(dev, true)**; /\*/drivers/base/dd.c\*/

}

\_\_device\_attach(dev, true)函数用于从总线探测匹配的驱动，函数定义在/drivers/base/dd.c文件内：

static int \_\_device\_attach(struct device \*dev, bool allow\_async)

/\*allow\_async：true\*/

{

int ret = 0;

device\_lock(dev); /\*锁定设备\*/

if (dev->driver) { /\*若设备已经关联了驱动device\_drvier实例\*/

if (klist\_node\_attached(&dev->p->knode\_driver)) {

/\*如果dev->p已经添加到device\_drvier中链表，返回1\*/

ret = 1;

goto out\_unlock;

}

/\*设备还没有绑定驱动\*/

ret = **device\_bind\_driver(dev)**; /\*主要完成设备与驱动绑定，/drivers/base/dd.c\*/

if (ret == 0)

ret = 1; /\*绑定成功\*/

else {

dev->driver = NULL; /\*绑定不成功\*/

ret = 0;

}

}

else { /\*尚未关联驱动\*/

struct device\_attach\_data data = { /\*/drivers/base/dd.c\*/

.dev = dev,

.check\_async = allow\_async,

.want\_async = false,

};

ret = **bus\_for\_each\_drv(dev->bus, NULL, &data,\_\_device\_attach\_driver)**;

/\*扫描总线上驱动实例，调用\_\_device\_attach\_driver()检查驱动是否匹配\*/

if (!ret && allow\_async && data.have\_async) {

dev\_dbg(dev, "scheduling asynchronous probe\n");

get\_device(dev);

async\_schedule(\_\_device\_attach\_async\_helper, dev);

} else {

pm\_request\_idle(dev);

}

}

out\_unlock:

device\_unlock(dev);

return ret; /\*成功返回0\*/

}

\_\_device\_attach(dev, true)函数调用关系如下图所示：



函数首先判断device实例是否已经关联了device\_driver实例，如果是且已绑定驱动则函数返回，如果没有绑定则调用device\_bind\_driver(dev)函数绑定驱动，主要工作是创建设备与驱动之间的符号链接，以及将设备添加到驱动中的设备链表。

如果设务尚未关联驱动，则调用 bus\_for\_each\_drv()函数逐个扫描总线驱动链表中的device\_driver实例，调用\_\_device\_attach\_driver()函数检查驱动是否与设备匹配，若匹配则调用总线驱动的probe()函数，完成设备驱动程序的注册，\_\_device\_attach\_driver()函数返回0。

\_\_device\_attach\_driver()函数暂且放一下，因为后面注册驱动时要扫描总线中的设备链表，也要执行匹配操作，因此介绍完驱动之后再一并介绍。

#### 3动态创建设备

前面介绍的注册device实例的device\_register(dev)函数用于注册静态定义或已经创建好的device实例，内核还提供了动态创建并添加device实例的接口函数**device\_create()**。

device\_create()函数定义在/drivers/base/core.c文件内，代码如下：

struct device \*device\_create(struct class \***class**, struct device \*parent,dev\_t devt, void \*drvdata, \

const char \*fmt, ...)

/\*class：所属设备类，不能为空，parent：父设备，devt：设备号，**drvdata**：传递给驱动的数据\*/

{

va\_list vargs;

struct device \*dev;

va\_start(vargs, fmt);

**dev = device\_create\_vargs(class, parent, devt, drvdata, fmt, vargs);**  /\*/drivers/base/core.c\*/

va\_end(vargs);

return dev;

}

struct device \*device\_create\_vargs(struct class \*class, struct device \*parent, \

dev\_t devt, void \*drvdata, const char \*fmt,va\_list args)

{

return **device\_create\_groups\_vargs(class, parent, devt, drvdata, NULL, fmt, args);**

}

device\_create\_groups\_vargs()函数定义如下（/drivers/base/core.c）：

static struct device \*device\_create\_groups\_vargs(struct class \*class, struct device \*parent, \

dev\_t devt, void \*drvdata, const struct attribute\_group \*\*groups, const char \*fmt, va\_list args)

{

struct device \*dev = NULL;

int retval = -ENODEV;

if (class == NULL || IS\_ERR(class)) /\*必须指定设备类\*/

goto error;

**dev = kzalloc(sizeof(\*dev), GFP\_KERNEL);** /\*分配device实例\*/

... /\*错误处理\*/

**device\_initialize(dev)**; /\*初始化device实例，同device\_register()中调用的初始化函数\*/

**dev->devt = devt**; /\*赋予设备号\*/

dev->class = **class**; /\*设备类\*/

dev->parent = parent; /\*父设备\*/

dev->groups = groups; /\*属性组，这里为NULL\*/

dev->release = device\_create\_release; /\*释放device实例的函数\*/

**dev\_set\_drvdata(dev, drvdata)**; /\***dev->driver\_data = data**，/include/linux/device.h\*/

retval = **kobject\_set\_name\_vargs(&dev->kobj, fmt, args);** /\*设置device.kobj名称\*/

if (retval)

goto error;

retval = **device\_add(dev)**; /\*添加device实例，见上文\*/

if (retval)

goto error;

return dev;

...

}

device\_create()函数不难理解，函数内动态分配device实例，初始化后添加到内核。需要注意的是函数参数必须指定设备类class（class参数不能为NULL），drvdata参数指向的是传递给驱动程序的数据（结构体实例），将赋予dev->driver\_data成员。

新创建device实例并没有指定所属总线，也就是说添加设备时不会探测驱动，但会动态创建设备文件（devt主设备号不为0时，这是一个非常重要的功能）。如果内核已经加载了此设备的驱动程序（设备类中设备共用驱动），则可以正常地访问设备。

#### 4设备资源

通用驱动模型中由devres结构体表示设备资源，device结构体双链表成员devres\_head管理了设备拥有的资源devres实例。设备资源管理数据结构组织关系如下图所示：



设备资源devres结构体定义在/drivers/base/devres.c文件内：

struct devres {

struct devres\_node node; /\*devres\_node结构体成员\*/

unsigned long long data[]; /\*具体资源的数据结构\*/

};

struct devres\_node {

struct list\_head **entry**; /\*添加到device.devres\_head双链表\*/

dr\_release\_t **release**; /\*释放资源函数指针，/include/linux/device.h\*/

#ifdef CONFIG\_DEBUG\_DEVRES

...

#endif

};

devres结构体是一个通用的数据结构，可表示设备的各种资源，如中断等。data成员一个占位符，实际是表示具体资源的数据结构，如表示中断资源的irq\_devres结构体。

devres\_node结构体中entry成员用于添加到设备的资源双链表，release为释放资源的函数指针，函数原型定义如下：

typedef void (\*dr\_release\_t)(struct device \*dev, void \*res);

typedef int (\*dr\_match\_t)(struct device \*dev, void \*res, void \*match\_data); /\*查找资源时的匹配函数\*/

分配设备资源devres实例的接口函数定义如下（drivers/base/devres.c）：

void \* **devres\_alloc**(dr\_release\_t release, size\_t size, gfp\_t gfp)

/\*release：释放资源的函数指针，size：资源数据结构大小，gfp：分配掩码\*/

{

struct devres \*dr;

dr = **alloc\_dr(release, size, gfp | \_\_GFP\_ZERO**);

/\*分配devres实例，后接size大小的表示具体资源结构体大小\*/

if (unlikely(!dr))

return NULL;

return **dr->data**; /\*返回表示具体资源的数据结构指针\*/

}

devres\_alloc()函数参数release表示释放资源的函数指针（函数需由具体资源的代码实现），size是具体资源数据结构大小。

alloc\_dr()函数用于分配devres结构体实例，其后紧接具体资源数据结构实例。devres\_alloc()函数返回具体资源数据结构实例指针，即dr->data。

分配资源devres实例后，需要调用接口函数**devres\_add(struct device \*dev, void \*res)**将其添加到设备device实例资源双链表的末尾，见/drivers/base/devres.c文件。

内核在/drivers/base/devres.c文件内还提供了管理（操作）设备资源的接口函数，例如：

●void \* devm\_kmalloc(struct device \*dev, size\_t size, gfp\_t gfp)：为设备分配并添加内存资源，大小为size，返回内存指针。设备解绑驱动时会自动释放内存资源。

●void \* devres\_find(struct device \*dev, dr\_release\_t release,dr\_match\_t match, void \*match\_data)：查找设备资源，返回资源指针，若不存在返回NULL。

●void \* devres\_get(struct device \*dev, void \*new\_res,dr\_match\_t match, void \*match\_data)：获取设备资源，若不存在则创建，返回资源指针。

●int devres\_release(struct device \*dev, dr\_release\_t release,dr\_match\_t match, void \*match\_data)：查找并释放资源devres，match()为在设备资源双链表中查找资源的匹配函数（match\_data为此函数参数），此参数可选。

●int devres\_release\_all(struct device \*dev)：释放设备持有的所有资源。

### 8.4.3驱动

驱动可以认为是驱动程序和设备的管理者。驱动由device\_driver结构体表示，当设备与驱动匹配时，将会调用device\_driver实例中的probe()函数完成设备驱动程序的加载。device\_driver结构体还定义了设备移出、睡眠、唤醒时的回调函数。device\_driver实例的管理比device实例要简单一些，device\_driver实例只挂接到总线bus\_type的驱动链表下。

在内核启动或加载模块时，将向内核注册device\_driver实例，注册时将驱动添加到总线上的驱动链表，并在总线上查找匹配的设备，匹配成功则加载设备驱动程序。

#### 1数据结构

驱动device\_driver结构体定义在/include/linux/device.h头文件：

struct device\_driver {

const char \*name; /\*驱动名称，匹配设备时，可能与设备名称进行匹配\*/

struct bus\_type **\*bus**; /\*所属总线\*/

struct module \*owner; /\*模块指针\*/

const char \*mod\_name; /\*模块名称\*/

bool **suppress\_bind\_attrs**; / \*禁止通过sysfs文件系统绑定/解绑\*/

enum  **probe\_type probe\_type**; /\*探测设备的方式，同步或异步\*/

const struct of\_device\_id **\*of\_match\_table;** /\*匹配列表，用于匹配设备树中设备节点\*/

const struct acpi\_device\_id \*acpi\_match\_table; /\*ACPI匹配列表\*/

**int (\*probe) (struct device \*dev)**; /\*探测设备函数（加载设备驱动程序）\*/

int (\*remove) (struct device \*dev); /\*解绑设备时调用\*/

void (\*shutdown) (struct device \*dev); /\*关机时用于退出设备\*/

int (\*suspend) (struct device \*dev, pm\_message\_t state); /\*使设备进入睡眠时调用（低功耗）\*/

int (\*resume) (struct device \*dev); /\*从睡眠模式唤醒设备时调用\*/

const struct attribute\_group \*\***groups**; /\*驱动属性\*/

**const struct dev\_pm\_ops \*pm**; /\*电源管理操作结构指针，/include/linux/pm.h\*/

struct driver\_private **\*p;**  /\*驱动私有数据结构指针\*/

};

device\_driver结构体主要成员简介如下：

●**name**：驱动名称，在匹配设备时，通常匹配操作最后通过此名称与设备名称比对（是否相同）。

●**of\_match\_table：**指向of\_device\_id结构体数组，用于匹配设备树中的设备节点。of\_device\_id结构体定义如下（/include/linux/mod\_devicetable.h）：

struct of\_device\_id {

char name[32];

char type[32];

char **compatible[128]**; /\*兼容属性\*/

const void \*data;

};

●acpi\_match\_table：指向acpi\_device\_id结构体，用于ACPI匹配列表。

●**pm：**指向设备电源管理dev\_pm\_ops结构体，定义如下（/include/linux/pm.h）：

struct dev\_pm\_ops {

int (\*prepare)(struct device \*dev);

void (\*complete)(struct device \*dev);

int (\*suspend)(struct device \*dev);

int (\*resume)(struct device \*dev);

int (\*freeze)(struct device \*dev);

int (\*thaw)(struct device \*dev);

int (\*poweroff)(struct device \*dev);

int (\*restore)(struct device \*dev);

int (\*suspend\_late)(struct device \*dev);

int (\*resume\_early)(struct device \*dev);

int (\*freeze\_late)(struct device \*dev);

int (\*thaw\_early)(struct device \*dev);

int (\*poweroff\_late)(struct device \*dev);

int (\*restore\_early)(struct device \*dev);

int (\*suspend\_noirq)(struct device \*dev);

int (\*resume\_noirq)(struct device \*dev);

int (\*freeze\_noirq)(struct device \*dev);

int (\*thaw\_noirq)(struct device \*dev);

int (\*poweroff\_noirq)(struct device \*dev);

int (\*restore\_noirq)(struct device \*dev);

int (\*runtime\_suspend)(struct device \*dev);

int (\*runtime\_resume)(struct device \*dev);

int (\*runtime\_idle)(struct device \*dev);

};

●**prob()**：探测函数指针，非常重要的函数，在设备与驱动匹配上时将会调用此函数加载设备驱动程序。

**●probe\_type：**probe\_type枚举类型成员，列举了驱动注册时探测设备的方式，枚举值定义如下：

enum probe\_type {

**PROBE\_DEFAULT\_STRATEGY**, /\*创建设备时的默认值\*/

PROBE\_PREFER\_ASYNCHRONOUS,

PROBE\_FORCE\_SYNCHRONOUS,

};

**●p：**驱动私有数据结构driver\_private指针，结构体定义在/drivers/base/base.h头文件：

struct driver\_private {

**struct kobject kobj;**  /\*跟踪device\_driver实例的kobject实例\*/

struct klist **klist\_devices**; /\*链接可驱动设备的device实例\*/

struct klist\_node **knode\_bus**; /\*将实例添加到总线的驱动链表\*/

struct module\_kobject \*mkobj;

struct device\_driver \*driver; /\*指向device\_driver实例\*/

};

device\_driver实例的管理比device实例要简单，它只添加到总线的驱动链表中，如下图所示：



#### 2注册驱动

注册驱动与注册设备相比要简单许多，主要工作如下：

（1）将device\_driver实例添加到总线的驱动链表，在总线对应目录项的drivers目录项下创建驱动对应的目录项。

（2）向驱动对应的目录项下添加驱动属性文件。

（3）匹配设备，匹配上则绑定设备到驱动，加载驱动程序。

注册device\_driver实例在sysfs中创建的目录项和文件如下图所示：



注册驱动的**driver\_register**(struct device\_driver \*drv)函数定义在/drivers/base/driver.c文件内：

int driver\_register(struct device\_driver \*drv)

{

int ret;

struct device\_driver \*other;

BUG\_ON(!drv->bus->p);

if ((drv->bus->probe && drv->probe) || (drv->bus->remove && drv->remove) || \

(drv->bus->shutdown && drv->shutdown))

... /\*输出警告信息，总线中定义的函数有优先权\*/

other = **driver\_find(drv->name, drv->bus)**;

/\*在总线中以驱动名称查找同名驱动是否存在，存在返回驱动指针，否则返回NULL\*、

if (other) {

printk(KERN\_ERR "Error: Driver '%s' is already registered, ""aborting...\n", drv->name);

return -EBUSY; /\*已经存在同名驱动，返回错误码\*/

}

ret = **bus\_add\_driver(drv)**; /\*将驱动添加到总线，/drivers/base/bus.c\*/

...

ret = **driver\_add\_groups(drv, drv->groups)**; /\*向驱动添加属性（数组），/drivers/base/driver.c\*/

...

**kobject\_uevent(&drv->p->kobj, KOBJ\_ADD)**; /\*最后触发uevent事件\*/

return ret;

}

注册函数首先在总线驱动链表中查找是否已经存在同名驱动，如果已经存在则返回错误码，否则调用函数**bus\_add\_driver(drv)**向总线添加驱动，然后添加驱动中定义的属性组，最后调用kobject\_uevent()函数向用户空间发送uevent事件。

下面看一下向总线添加驱动的bus\_add\_driver()函数实现，代码如下（/drivers/base/bus.c）：

int bus\_add\_driver(struct device\_driver \*drv)

{

struct bus\_type \*bus;

struct driver\_private \*priv;

int error = 0;

bus = bus\_get(drv->bus); /\*获取驱动所属总线实例，增加总线引用计数\*/

...

priv = **kzalloc(sizeof(\*priv), GFP\_KERNEL)**; /\*分配driver\_private实例\*/

...

klist\_init(&priv->klist\_devices, NULL, NULL); /\*初始化设备链表头\*/

**priv->driver = drv**;

**drv->p = priv**;

**priv->kobj.kset = bus->p->drivers\_kset**; /\*kobjedct集合，父目录项\*/

error = **kobject\_init\_and\_add(&priv->kobj, &driver\_ktype, NULL, "%s", drv->name)**;

/\*添加priv->kobj，kobject类型为**driver\_ktype**\*/

...

**klist\_add\_tail(&priv->knode\_bus, &bus->p->klist\_drivers)**;

/\*driver\_private添加到总线驱动链表末尾\*/

if (**drv->bus->p->drivers\_autoprobe**) { /\*允许自动探测设备，总线默认允许\*/

if (driver\_allows\_async\_probing(drv)) { /\*判断是否异步探测，/drivers/base/dd.c\*/

pr\_debug("bus: '%s': probing driver %s asynchronously\n",drv->bus->name, drv->name);

async\_schedule(driver\_attach\_async, drv); /\*异步绑定设备，/kernel/async.c\*/

}

else {

error = **driver\_attach(drv)**; /\*探测匹配设备，/drivers/base/dd.c\*/

...

}

}

module\_add\_driver(drv->owner, drv);

error = **driver\_create\_file(drv, &driver\_attr\_uevent)**; /\*添加uevent属性\*/

...

error = **driver\_add\_groups(drv, bus->drv\_groups)**; /\*添加总线默认的驱动属性\*/

...

if (!**drv->suppress\_bind\_attrs**) {

error = **add\_bind\_files(drv)**; /\*添加unbind、bind属性，/drivers/base/bus.c\*/

...

}

return 0;

...

}

bus\_add\_driver()函数主要工作是为驱动创建driver\_private实例并初始化，将driver\_private添加到总线的驱动链表中，并添加priv->kobj实例，在总线drivers目录项下创建驱动的目录项；向驱动目录项下添加驱动属性文件；探测匹配的设备。

##### ■驱动属性

驱动属性由driver\_attribute结构体表示，定义如下（/include/linux/device.h）：

struct driver\_attribute {

struct attribute attr; /\*通用属性\*/

ssize\_t (\*show) (struct device\_driver \*driver, char \*buf); /\*读特定属性函数\*/

ssize\_t (\*store) (struct device\_driver \*driver, const char \*buf,size\_t count); /\*写特定属性函数\*/

};

在同一头文件内还给出了定义驱动属性的宏，例如：

#define DRIVER\_ATTR(\_name, \_mode, \_show, \_store) \

struct driver\_attribute driver\_attr\_##\_name = \_\_ATTR(\_name, \_mode, \_show, \_store)

#define DRIVER\_ATTR\_RW(\_name) \

struct driver\_attribute driver\_attr\_##\_name = \_\_ATTR\_RW(\_name)

在前面介绍的添加priv->kobj实例函数中，实例kobj->ktype赋值为**driver\_ktype**实例指针，其中包含通用读写驱动属性文件的函数。

**driver\_ktype**实例定义在/drivers/base/bus.c文件内：

static struct kobj\_type driver\_ktype = {

.sysfs\_ops = &**driver\_sysfs\_ops**,

.release = driver\_release,

};

driver\_sysfs\_ops实例定义如下：

static const struct sysfs\_ops driver\_sysfs\_ops = {

.show = drv\_attr\_show, /\*读驱动属性函数\*/

.store = drv\_attr\_store, /\*写驱动属性函数\*/

};

通用驱动读写函数与之前介绍的属性读写函数相似，最终调用具体driver\_attribute实例内定义的读写函数读写属性，如下图所示。



##### ■匹配设备

在向总线添加驱动的bus\_add\_driver()函数中，将扫描总线下的设备链表，查找匹配的设备，匹配成功则将设备绑定驱动，加载驱动程序，此项工作通常由**driver\_attach(drv)**函数完成。

driver\_attach()函数定义如下（/drivers/base/dd.c）：

int driver\_attach(struct device\_driver \*drv)

{

return **bus\_for\_each\_dev**(drv->bus, NULL, drv, **\_\_driver\_attach**); /\*/drivers/base/bus.c\*/

}

bus\_for\_each\_dev()函数扫描总线中的设备链表，对每个设备调用\_\_driver\_attach(dev, data)函数判断设备是否与驱动匹配，若匹配则调用探测函数向内核注册设备驱动程序，这与注册设备时扫描驱动链表类似。下一小节将专门介绍设备与驱动的匹配。

### 8.4.4匹配与探测

在注册设备时将扫描总线的驱动链表并调用\_\_device\_attach\_driver(drv,dev)函数判断当前驱动与设备是否匹配；在注册驱动时，将扫描总线中的设备链表，对每个设备调用\_\_driver\_attach(dev, drv)函数，检查设备与驱动是否匹配，函数调用关系如下图所示。



匹配与探测主要分两步，一是调用总线的match()函数检查设备与驱动是否匹配，若匹配则进行第二步，二是调用总线或驱动中定义的probe()函数，加载设备驱动程序。

\_\_device\_attach\_driver(drv,dev)和\_\_driver\_attach(dev, drv)函数都定义在/drivers/base/dd.c文件内，代码如下：

static int \_\_device\_attach\_driver(struct device\_driver \*drv, void \*\_data)

{

struct device\_attach\_data \*data = \_data;

**struct device \*dev = data->dev**; /\*设备device实例\*/

bool async\_allowed;

if (dev->driver) /\*设备已经关联了驱动，返回错误码\*/

return -EBUSY;

if (!**driver\_match\_device(drv, dev)**) /\*判断设备与驱动是否匹配，/drivers/base/base.h\*/

return 0; /\*不匹配返回0\*/

async\_allowed = driver\_allows\_async\_probing(drv);

if (async\_allowed)

data->have\_async = true;

if (data->check\_async && async\_allowed != data->want\_async)

return 0;

return **driver\_probe\_device(drv, dev)**; /\*调用总线或驱动probe()函数，/drivers/base/dd.c\*/

}

static int \_\_driver\_attach(struct device \*dev, void \*data)

{

struct **device\_driver \*drv = data**;

if (!**driver\_match\_device(drv, dev)**) /\*判断设备与驱动是否匹配，/drivers/base/base.h\*/

return 0; /\*不匹配返回0\*/

if (dev->parent) /\* Needed for USB \*/

device\_lock(dev->parent);

device\_lock(dev);

if (!dev->driver)

**driver\_probe\_device(drv, dev)**; /\*调用总线或驱动probe()函数，/drivers/base/dd.c\*/

device\_unlock(dev);

if (dev->parent)

device\_unlock(dev->parent);

return 0;

}

#### 1匹配函数

driver\_match\_device(drv, dev)函数定义在/drivers/base/base.h头文件内，用于检查驱动与设备是否匹配：

static inline int driver\_match\_device(struct device\_driver \*drv, struct device \*dev)

{

return drv->bus->match ? **drv->bus->match(dev, drv)** : 1; /\*调用总线匹配函数，否则返回1\*/

}

如果总线定义了match()函数，则调用它检查是否匹配，否则返回1（默认匹配）。函数返回非零值表示匹配，返回0表示不匹配。

#### 2探测函数

若驱动与设备匹配，将继续调用driver\_probe\_device(drv, dev)函数绑定设备与驱动，函数内调用总线或驱动的probe()函数。

driver\_probe\_device()函数定义如下（/drivers/base/dd.c）：

int driver\_probe\_device(struct device\_driver \*drv, struct device \*dev)

{

int ret = 0;

if (!**device\_is\_registered(dev)**) /\*device是否已注册（导出到sysfs），/include/linux/device.h\*/

return -ENODEV;

... /\*输出信息\*/

pm\_runtime\_barrier(dev);

**ret = really\_probe(dev, drv)**; /\*/drivers/base/dd.c\*/

**pm\_request\_idle(dev)**;

return ret;

}

really\_probe(dev, drv)函数定义如下：

static int really\_probe(struct device \*dev, struct device\_driver \*drv)

{

int ret = 0;

int local\_trigger\_count = atomic\_read(&deferred\_trigger\_count);

atomic\_inc(&probe\_count);

... /\*输出信息\*/

WARN\_ON(!list\_empty(&dev->devres\_head));

**dev->driver = drv**; /\*关联驱动\*/

ret = pinctrl\_bind\_pins(dev); /\*pinctrl子系统，/drivers/base/pinctrl.c\*/

if (ret)

goto probe\_failed;

if (**driver\_sysfs\_add(dev)**) { /\*创建符号链接等，/drivers/base/dd.c\*\*/

... /\*输出信息\*/

goto probe\_failed;

}

if (dev->pm\_domain && dev->pm\_domain->activate) {

ret = d**ev->pm\_domain->activate(dev)**; /\*pm\_domain中激活函数\*/

...

}

if (**dev->bus->probe**) { /\*优先调用总线的probe()函数\*/

**ret = dev->bus->probe(dev)**;

...

} else if (**drv->probe**) { /\*总线没有定义probe()函数，则调用驱动的probe()函数\*/

**ret = drv->probe(dev)**;

...

}

if (dev->pm\_domain && dev->pm\_domain->sync)

dev->pm\_domain->sync(dev); /\*调用pm\_domain中同步函数\*/

**driver\_bound(dev)**; /\*绑定设备与驱动，设备添加到驱动中链表等，/drivers/base/dd.c\*/

ret = 1;

... /\*输出信息\*/

goto done;

probe\_failed:

... /\*输出信息\*/

done:

atomic\_dec(&probe\_count);

**wake\_up(&probe\_waitqueue)**; /\*唤醒在probe\_waitqueue队列睡眠等待进程\*/

return ret;

}

really\_probe()函数中将关联设备与驱动，如果总线bus\_type实例定义了probe()函数，则调用总线定义的probe()函数，否则调用驱动定义的probe()函数。在probe()函数中，将完成设备驱动程序的加载，即设备驱动数据结构（cdev或gendisk）实例的定义和注册。

really\_probe()函数中调用driver\_sysfs\_add()函数创建驱动与设备在sysfs中的符号链接，函数定义如下：

static int driver\_sysfs\_add(struct device \*dev)

{

int ret;

if (dev->bus) /\*执行通知链\*/

blocking\_notifier\_call\_chain(&dev->bus->p->bus\_notifier,BUS\_NOTIFY\_BIND\_DRIVER, dev);

ret = sysfs\_create\_link(&dev->driver->p->kobj, &dev->kobj,kobject\_name(&dev->kobj));

/\*在驱动目录项下创建到设备的符号链接，名称为设备名称\*/

if (ret == 0) {

ret = sysfs\_create\_link(&dev->kobj, &dev->driver->p->kobj,"driver");

/\*在设备目录项下创建到驱动的符号链接，名称为“driver”\*/

if (ret)

sysfs\_remove\_link(&dev->driver->p->kobj,kobject\_name(&dev->kobj));

}

return ret;

}

设备device结构体和驱动device\_driver结构体通常嵌入到具体总线定义的设备和驱动数据结构中。设备（device实例）可以在平台（板级）文件中定义，并在初始化函数中向内核注册。驱动在设备驱动程序文件中定义，在初始化函数或加载模块时，调用注册函数向内核注册。

在平台（板级）代码中定义并注册设备实例，会产生很多“垃圾”代码，并且每次修改都需要重新编译内核，更现代的方法是通过设备树向内核传递设备硬件信息，详见下节。

## 8.5设备树

在通用驱动模型中，每个外部设备需要由device结构体（或封装它的数据结构）表示，并向内核注册。这些代码位于体系结构相关的平台（板级）文件中（例如：/arch/mips/loongson32/common/platform.c），另外还需要定义描述硬件信息的数据结构实例，并关联到表示设备的数据结构中，以便驱动程序获取硬件信息，如控制寄存器地址、中断号等。

由于Linux内核支持的体系结构、平台众多，以及外部设备类型的繁杂，导致在内核中描述硬件信息的代码数量众多，而这些代码对内核来说是没有什么技术含量的“垃圾”代码，为此内核采用了设备树，将描述硬件信息的代码从内核代码中分离出来。

设备树是用于描述系统硬件信息的文件，分为源文件和目标文件。源文件（DTS，.dts）相当于C语言的源文件，它用规定的语法格式描述系统硬件信息，通过设备树文件编译器（DTC）编译后生成设备树目标文件（DTB，.dtb），相当于C语言目标文件，这是个二进制文件。

设备树目标文件可以先编译好（不编入内核），保存在固件中，然后传递给内核，内核即可从设备树目标文件中获取硬件信息。设备树源文件也可以存放在内核源文件中，连同内核一起编译（内核提供设备树编译器），并将目标文件嵌入到内核可执行文件中。内核在启动阶段，将从设备树目标文件中获取设备信息，并在内核中用device\_node实例（相当于硬件设备库）表示硬件信息，设备驱动程序可从此数据结构中获取指定硬件的信息，用其创建设备实例，并向内核注册。

设备树说明文档《Devicetree Specifification》可从https://www.devicetree.org/specifications/网站下载。

### 8.5.1设备树源文件

设备树源文件位于体系结构相关的/arch/mips/boot/bts/目录下，设备树源文件由设备节点组成，每个设备对应源文件中一个节点，设备节点组成父子关系的层次结构。设备节点由节点名称（设备名称）和属性组成。属性由形如“property=value”的属性名称和属性值组成，"property"为属性名称，"value"为属性值。属性包含设备兼容属性、寄存器属性、中断号属性等。

下面用一个简单的例子来演示设备树源文件的组成：

/{ //根节点，名称为“/”

compatible = "nvidia,harmony", "nvidia,tegra20"; //平台兼容属性

#address-cells = <1>;

#size-cells = <1>;

interrupt-parent = <&intc>; //中断控制器节点

chosen { }; //chosen节点，传递命令行参数等

aliases { }; //别名节点

memory { //物理内存节点

device\_type = "memory";

reg = <0x00000000 0x40000000>; //物理内存起始地址和长度

};

soc { //soc节点开始

compatible = "nvidia,tegra20-soc", "simple-bus";

#address-cells = <1>;

#size-cells = <1>;

ranges;

intc: interrupt-controller@50041000 { //中断控制器，intc为标号，等同节点地址

compatible = "nvidia,tegra20-gic";

interrupt-controller; //标记为中断控制器

#interrupt-cells = <1>;

reg = <0x50041000 0x1000>, < 0x50040100 0x0100 >; //控制寄存器起始地址和长度

};

serial@70006300 { //串口设备节点

compatible = "nvidia,tegra20-uart";

reg = <0x70006300 0x100>;

interrupts = <122>; //中断号

};

i2s1: i2s@70002800 {

compatible = "nvidia,tegra20-i2s";

reg = <0x70002800 0x100>;

interrupts = <77>;

codec = <&wm8903>;

};

i2c@7000c000 {

compatible = "nvidia,tegra20-i2c";

#address-cells = <1>;

#size-cells = <0>;

reg = <0x7000c000 0x100>;

interrupts = <70>;

wm8903: codec@1a {

compatible = "wlf,wm8903";

reg = <0x1a>;

interrupts = <347>;

};

}; //i2c@7000c000节点结束

}; //soc节点结束

sound { //sound节点

compatible = "nvidia,harmony-sound";

i2s-controller = <&i2s1>;

i2s-codec = <&wm8903>;

};

};

设备树源文件中各节点组成父子层次结构，必须有一个根节点，名称为“/”，它是层次结构中最顶层节点。根节点下有子节点，如上所示，根节点下存在chosen、aliases、memory、soc等子节点，而soc节点下又存在子节点 interrupt-controller@50041000、i2c@7000c000等。

节点内容是形如“property=value”的属性、值列表，表示硬件的具体信息。下文中将介绍设备树源文件中节点和属性相关的内容。

#### 1节点

设备树源文件中节点名称一般形式为：

node-name@unit-address

node-name为设备名称，最长为31个字符，第一个字符需是字母，字符可以是数字和字母，以及有限的几个其它字符，详见说明文档。设备名称通常表示设备类型，如cpu、memory、i2c等。

unit-address也由字符组成，表示设备在总线上的位置（地址），它必须与该节点中reg属性中第一个地址值相同，例如：

serial@**70006300** { //串口设备节点，70006300 前没有加0x

compatible = "nvidia,tegra20-uart";

reg = <**0x70006300** 0x100>; //起始地址为**0x70006300**

interrupts = <122>; //中断号

};

如果节点属性中不存在reg属性，则节点名称中@unit-address字符必须省略，只剩node-name字符。

推荐使用的节点名称如下（详见说明文档）：

• adc

• accelerometer

• atm

• audio-codec

• audio-controller

• backlight

• bluetooth

• bus

• cache-controller

• camera

• can

• charger

• clock

• clock-controller

• compact-flflash

• cpu

• cpus

• crypto

• disk

...

设备树源文件中的节点可以用路径来表示，路径由从根节点到本节点的节点名称组成，中间用‘/’字符隔开，类似文件路径。例如：上例中i2c@7000c000节点的路径为/soc/i2c@7000c000。

#### 2属性

节点下的内容是属性，用于描述硬件的物理信息，属性通常具有值，但是值也可以为空。属性名称与节点名称类似，最多也是由31个字符组成。属性值由0个或多个字节组成，4个字节（32位）表示一个单元（cell）。

常用的属性值有以下几种：

●**空**：不需要等号和属性值，只需要属性名称，通常用于bool量，属性名称存在代表true，属性名称不存在代表false。例如：interrupt-controller；（表示设备节点是中断控制器）

●**32位整数**：整数值在目标文件中以大端模式存放，整数值放在< >尖括号内。例如：interrupts = <122>表示设备中断编号为122。

●**64位整数：**与32位整数类似，也是大端存放。

●**字符串**：与C语言中字符串相似，以‘\0’字符结束，例如：compatible = "arm,cortex-a9"，字符串放在双引号内。

●**字符串列表**：属性值由多个字符串组成，例如：string-list = "red fish", "blue fish"；

●**节点地址**：实际为32位（无符号）整数，表示节点在目标文件中的地址，用于节点引用。例如：i2s-controller = <&i2s1>;表示中断控制器节点的地址，i2s1为中断控制器节点标号。在节点名称前面可以加标号（类似C语言中标号）表示节点的地址。

下面简要介绍几个常用标准属性及其属性值类型，更多的属性请读者参考说明文档：

●**compatible：**兼容属性，属性值为字符串或字符串列表，表示节点与设备驱动的匹配属性。属性值推荐格式为"manufacturer,model"，前者表示设备生产商，后者表示型号。例如：compatible = "fsl,mpc8641";

●**model**：model属性与compatible属性类似，但其属性值只能是字符串，而不是字符串列表。

●**phandle**：句柄属性，属性值为节点编号，用于其它节点引用本节点，例如：phandle = <1>;

●**status**：状态属性，属性值为字符串，属性值取值："okay" 、"disabled" 、"reserved" 、"fail"、"fail-sss" 。

●**#address-cells, #size-cells**：#address-cells表示reg属性值中地址值由多少个cell表示，如果地址是32位，则属性值为1。#size-cells表示reg属性值中长度值由多少个cell表示，如果长度32位值，则属性值为1。这里表示的reg属性是指当前节点的子节点中的属性，而不是本节点reg属性（本节点的在父节点中指定）。

●**reg**：地址区间属性，属性值为<address length address length...>，每个address length对表示一个地址区间，address表示起始地址，length表示长度。前面介绍的#address-cells和 #size-cells属性表示address和length分别由多少个cell（32位）表示。例如：

#address-cells = <1>; //32位地址

#size-cells = <1>; //32位长度值

memory { //物理内存节点

device\_type = "memory";

reg = <0x00000000 0x40000000>; //物理内存起始地址和长度值都为32位

};

●**interrupts：**中断属性，属性值通常表示设备产生的中断编号，例如：

interrupts = <0xA>;

●**#interrupt-cells：**中断控制器interrupt-controller节点中的属性，表示描述中断域需要的cell数量。

●**interrupt-controller：**中断控制器interrupt-controller节点中的属性，属性值为空，存在此属性则表示节点为中断控制器。

#### 3常用节点及属性

下面简要介绍几个设备树源文件中常用的节点及其属性，更详细的信息请参考说明文档。

●**根节点**

根节点为‘/’，必须的属性如下：

#address-cells

#size-cells

model

compatible

●**aliases**

别名节点，通常是根节点下的子节点。节点内属性用于给其它节点取别名，例如：

aliases {

serial0 = "/simple-bus@fe000000/serial@llc500"; //serial0代表serial@llc500节点

ethernet0 = "/simple-bus@fe000000/ethernet@31c000";

};

●**memory**

物理内存节点，表示物理内存起始地址及长度。例如：

#address-cells = <2> ; //起始地址为64位

#size-cells = <2>; //长度值为64位

memory@0 {

reg = <0x000000000 0x00000000 0x00000000 0x80000000 //64位，高位在前

0x000000001 0x00000000 0x00000001 0x00000000>;

};

上面表示的物理内存包含两段分别是：起始地址0x0长度0x0x80000000（2GB）、起始地址0x1 00000000，长度0x1 00000000 (4 GB)。

●**chosen**

chosen节点并不表示设备，其属性值表示命令行参数等信息，通常是根节点下的子节点。例如：

chosen {

**bootargs** = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200"; //命令行参数

};

**●cpus/cpu**

cpus表示系统中CPU核的共用信息，其子节点cpu表示某一个CPU核的信息。cpus节点常用属性有：

#address-cells //表示子节点cpu中reg属性起始地址

#size-cells //属性值通常为0

cpu节点为cpus节点的子节点，表示CPU核的物理信息，常用属性请参考设备树说明文档。下面看一个简单的例子：

cpus { //cpus节点下存在一个cpu子节点

#address-cells = <1>;

#size-cells = <0>;

cpu@0 {

reg = <0>; //提供给线程的CPU核编号

d-cache-block-size = <32>; // L1数据缓存块大小32 bytes

i-cache-block-size = <32>; // L1指令缓存块大小32 bytes

d-cache-size = <0x8000>; // L1数据缓存大小32K

i-cache-size = <0x8000>; // L1指令缓存块大小32K

timebase-frequency = <82500000>; // 时钟计数器频率82.5 MHz

clock-frequency = <825000000>; // CPU核时钟频率825 MHz

};

};

另外cpu节点中还需要包含TLB信息等，详情请参考说明文档。

设备树源文件中需要注明源文件采用的DTS版本号，否则默认为版本0。源文件也可以像C源文件一样通过头文件引入其它文件，例如：

/dts-v1/; //采用版本1

/include/ "bcm6328.dtsi" //bcm6328.dtsi必须与当前文件在同一目录下

内核在/Documentation/devicetree/bindings/目录下保存了设备节点的说明文档，例如属性值含义等。如果在设备树中增加新的设备节点名称，需要在此目录下添加说明文档。

### 8.5.2设备树目标文件

设备树源文件需经过设备树编译器DTC编译后生成具有规定格式的目标文件，后缀为.dtb，相当于C编译器输出的目标文件。本小节简要介绍设备树目标文件的结构以及如何将目标文件传递给内核。

#### 1目标文件格式

设备树目标文件格式如下图所示，目标文件主要包含以下部分：设备树目标文件头数据结构，节点结构区域，属性字符串区域，保留内存区域等。

free space是为满足对齐要求空闲的区域。目标文件加载到物理内存中时，文件开头位于低地址。



**●fdt\_header:**

目标文件头是一个fdt\_header结构体实例，结构体中成员都是32位整数（大端存放），结构体定义如下：

struct fdt\_header {

uint32\_t magic; /\*魔数，用于判定是否是设备树目标文件，0xd00dfeed\*/

uint32\_t totalsize; /\*目标文件总大小，字节数\*/

uint32\_t off\_dt\_struct; /\*structure block相对于文件头的偏移量，字节数\*/

uint32\_t off\_dt\_strings; /\*strings block相对文件头的偏移量，字节数\*/

uint32\_t off\_mem\_rsvmap; /\*memory reservation block相对文件头的偏移量，字节数\*/

uint32\_t version; /\*目标文件版本号\*/

uint32\_t last\_comp\_version; /\*最低兼容的文件版本号\*/

uint32\_t boot\_cpuid\_phys; /\*启动CPU核ID号\*/

uint32\_t size\_dt\_strings; /\*structure block部分大小，字节数\*/

uint32\_t size\_dt\_struct; /\*strings block部分大小，字节数\*/

};

**●memory reservation block:**

memory reservation block区域保存的是fdt\_reserve\_entry结构体实例数组，结构体定义如下：

struct fdt\_reserve\_entry {

fdt64\_t address; /\*64位地址，大端存放\*/

fdt64\_t size; /\*64位长度，大端存放\*/

};

fdt\_reserve\_entry结构体表示一段物理内存，起始地址和长度值都是64位，即使在32位系统中也一样，在32位系统中高32位被忽略。数组的最末尾项address和size成员值都为0。

数组项中表示的物理内存段是保留的物理内存，只能由引导加载程序访问，内核等用户程序不能访问。保留物理内存段在memory节点中设置。

**●structure block：**

structure block部分描述了设备树中节点信息，其结构如下图所示：



每个节点由以下部分组成（斜线阴影部分表示为满足对齐要求预留的空间）：

●FDT\_BEGIN\_NODE：节点开始标记，值为0x00000001（大端存放），后跟节点名称字符串，为满足对齐要求，可能预留有空间。

●FDT\_PROP：属性标记，值为0x00000003，表示一个属性的开始，后接以下结构体实例：

struct {

uint32\_t len; /\*属性值的长度，字节数\*/

uint32\_t nameoff; /\*属性名称指针，属性名称保存在strings block区域\*/

}

在结构体实例之后保存的是属性值。

●FDT\_NOP：空标记（可选），取值为0x00000004，没有实际的意义，表示删除节点。

●FDT\_END\_NODE：节点结束标记，取值0x00000002。

子节点与父节点的属性并列，子节点结构也如上所述，也就是说子节点是嵌套在父节点内部的。节点在structure block中是层层嵌套的。

structure block区域最后是FDT\_END (0x00000009)标记，标记structure block区域结束。

**●strings block:**

strings block区域保存的是设备树所有节点属性名称字符串，此区域没有对齐要求。

内核在/scripts/dtc/libfdt/目录下的文件内定义了设备树目标文件底层的操作接口函数，在/drivers/of/fdt.c文件内定义了更高层次的操作接口函数。例如：

●void \*\_\_init **of\_get\_flat\_dt\_prop**(unsigned long node, const char \*name,int \*size)：node表示目标文件加载到内存后节点地址，name表示属性名称字符串，函数返回属性值指针，函数调用后size保存属性值长度，字节数。

●nt \_\_init of\_flat\_dt\_is\_compatible(unsigned long node, const char \*compat)：node指向节点是否匹配参数compat指向的兼容属性。

#### 2链接目标文件

如果内核需要支持设备树，需要选择OF配置选项（/drivers/of/Kconfig），这个选项是在体系结构（板级）相关的配置文件中直接选择，而不是通过配置界面选择。例如，MIPS体系结构在/arch/mips/Kconfig配置文件中定义了设备树相关的配置选项：

config USE\_OF /\*支持设备树\*/

bool

select **OF**

select **OF\_EARLY\_FLATTREE /\*早期初始化调用\*/**

select IRQ\_DOMAIN

config **BUILTIN\_DTB** /\*使用/arch/mips/boot/dts/目录下设备树源文件，目标文件嵌入内核镜像文件\*/

bool

如果平台需要使用/arch/mips/boot/dts/目录下的设备树源文件，则需要选择USE\_OF和BUILTIN\_DTB配置选项，例如（/arch/mips/Kconfig）：

config CAVIUM\_OCTEON\_SOC /\*平台类型选项\*/

...

select USE\_OF

...

select BUILTIN\_DTB

...

如果选择了BUILTIN\_DTB配置选项，内核会在**/arch/mips/boot/dts/**目录下根据配置选项选择设备树源文件进行编译（自带DTC编译器），编译生成的设备树目标文件嵌入到内核镜像文件初始化数据段中。

在体系结构无关的头文件/include/asm-generic/vmlinux.lds.h中定义了设备树目标文件段：

#define KERNEL\_DTB() \

STRUCT\_ALIGN(); \

VMLINUX\_SYMBOL(**\_\_dtb\_start**) = .; \ /\*设备树目标文件起始地址\*/

\*(.dtb.init.rodata) \

VMLINUX\_SYMBOL(**\_\_dtb\_end**) = .; /\*设备树目标文件结束地址\*/

#define INIT\_DATA \ /\*初始化数据段\*/

\*(.init.data) \

...

**KERNEL\_DTB()**  \ /\*设备树目标文件\*/

...

另外也可以将设备树目标文件打包到内核镜像的末尾，需要选择相应的配置选项（/arch/mips/Kconfig）：

choice

prompt "Kernel appended dtb support" if OF

default MIPS\_NO\_APPENDED\_DTB /\*不将设备树目标文件与内核目标文件打包\*/

config MIPS\_NO\_APPENDED\_DTB

bool "None"

help

Do not enable appended dtb support.

config MIPS\_RAW\_APPENDED\_DTB /\*设备树目标文件打包到vmlinux.bin内核目标文件末尾\*/

bool "vmlinux.bin" /\*没有压缩的目标文件\*/

help

...

config MIPS\_ZBOOT\_APPENDED\_DTB

/\*设备树目标文件打包到vmlinuz.bin内核目标文件末尾\*/

bool "vmlinuz.bin" /\*压缩的目标文件\*/

depends on SYS\_SUPPORTS\_ZBOOT

help

...

endchoice

endmenu

在链接文件/arch/mips/kernel/vmlinux.lds.S中为设备树目标文件预留了空间：

SECTIONS

{

...

#ifdef CONFIG\_MIPS\_RAW\_APPENDED\_DTB /\*内核镜像的末尾\*/

**\_\_appended\_dtb** = .; /\*初始化完成后将释放此空间\*/

/\*为DTB预留空间\*/

. += 0x100000;

#endif

...

\_\_init\_end = .;

BSS\_SECTION(0, 0x10000, 0)

\_end = . ;

...

}

}

如果设备树目标文件由引导加载程序（固件）加载并传递给内核，则引导加载程在将设备树目标文件加载到内存后，需要将设备树目标文件地址写入a1通用寄存器传递给内核。

如果在内核镜像末尾拼接上设备树目标文件，则此设备树目标文件具有优先权（不使用引导加载程序传入的目标文件），其地址将写入a1寄存器（head.S）。在head.S源文件中，通用寄存器a1值将保存至全局变量fw\_arg1，表示命令行参数或设备树目标文件地址，如果使用的是打包的设备树目标文件，a0寄存器值将赋值-2。

内核是使用内嵌的设备树目标文件还是拼接在内核镜像末尾（或外部传入）的设备树目标文件以及优先顺序，由平台相关代码中的plat\_mem\_setup()函数确定。

### 8.5.3使用设备树

设备树目标文件可以由引导加载程序传入内核，也可以将设备树目标文件拼接在内核镜像末尾，也可以直接将目标文件嵌入到内核镜像初始化数据段中。

内核在启动阶段早期会从设备树目标文件中获取信息，例如：命令行参数等，随后将设备树目标文件中节点在内核展开，由device\_node结构体实例表示设备节点，最后还需要将节点信息导出到sysfs文件系统，函数调用关系如下图所示。



\_\_dt\_setup\_arch(void \*bph)函数用于在早期读取设备树中信息至内核，例如：命令行参数等，通常将读取信息保存至全局变量。

**unflatten\_and\_copy\_device\_tree()**函数用于将设备树目标文件在内核中展开成device\_node实例。

of\_core\_init()函数用于将设备树节点和属性信息导出至sysfs文件系统。

设备树在内核中最终由device\_node结构体实例组成的层次结构表示，内核提供了从层次结构中查找节点、读取属性值等接口函数，驱动程序可调用这些接口函数查找并获取设备的物理信息。

#### 1设备树初始化

内核在初始化阶段的早期需要从设备树文件中读取信息至内核全局变量中，函数调用关系如下：

static void \_\_init arch\_mem\_init(char \*\*cmdline\_p)

{

...

**plat\_mem\_setup();** /\*读取设备树中部分信息\*/

...

**device\_tree\_init()**; /\*设备树目标文件展开成device\_node结构体实例，平台（板级）实现\*/

...

}

驱动模型初始化函数中与设备树相关的函数如下：

void \_\_init driver\_init(void)

{

...

**of\_core\_init();** /\*将设备树节点信息导出到/sys/fireware/devicetree/，/drivers/of/base.c\*/

}

of\_core\_init()函数负责将设备树中节点信息导出到/sys/fireware/devicetree/目录下，节点表现为目录项，属性表现为文件，源代码请读者自行阅读。

##### ■早期初始化

在平台（板级）实现的 **plat\_mem\_setup()**函数中需要调用\_\_dt\_setup\_arch(void \*bph)函数读取设备树中信息，参数bph表示设备树目标文件地址。参数bph值可以是fw\_arg1或\_\_dtb\_start，以确定使用哪个设备树目标文件。

\_\_dt\_setup\_arch(void \*bph)函数定义如下（/arch/mips/kernel/prom.c体系结构相关的函数）：

void \_\_init \_\_dt\_setup\_arch(void \*bph) /\*bph表示设备树目标文件地址\*/

{

if (!**early\_init\_dt\_scan(bph)**) /\*/drivers/of/fdt.c\*/

return;

mips\_set\_machine\_name(of\_flat\_dt\_get\_machine\_name()); /\*从设备树中读取机器名称\*/

}

early\_init\_dt\_scan()函数定义如下：

bool \_\_init early\_init\_dt\_scan(void \*params) /\*params为设备树目标文件地址\*/

{

bool status;

status = **early\_init\_dt\_verify(params)**;

/\*设备树目标文件地址保存至全局变量initial\_boot\_params，/drivers/of/fdt.c\*/

if (!status)

return false;

**early\_init\_dt\_scan\_nodes()**; /\*读取设备树部分节点属性信息，/drivers/of/fdt.c\*/

return true;

}

early\_init\_dt\_verify(params)函数检查设备树目标文件的有效性后，将设备树目标文件地址赋予全局变量**initial\_boot\_params**，后面的函数中将使此变量访问设备树目标文件。

early\_init\_dt\_scan\_nodes()函数用于读取设备树中信息，函数定义如下（体系结构无关的函数）：

void \_\_init early\_init\_dt\_scan\_nodes(void)

{

/\*从 /chosen节点读取命令行参数复制到boot\_command\_line指向空间\*/

of\_scan\_flat\_dt(**early\_init\_dt\_scan\_chosen**, boot\_command\_line); /\*/drivers/of/fdt.c\*/

/\*读取根节点"#address-cells"和"#size-cells"属性值，保存至全局变量\*/

of\_scan\_flat\_dt(**early\_init\_dt\_scan\_root**, NULL); /\*/drivers/of/fdt.c\*/

/\*读取物理内存信息，调用memblock\_add(base, size)函数，memory节点必须定义设备类型属性\*/

of\_scan\_flat\_dt(**early\_init\_dt\_scan\_memory**, NULL); /\*/drivers/of/fdt.c\*/

}

of\_scan\_flat\_dt(int (\*it)(),void \*data)函数用于扫描设备树中所有节点，对每个节点都调用**it()**函数。

##### ■展开设备树

在体系结构相关的arch\_mem\_init()函数中，在初始化自举分配器后将调用**device\_tree\_init()**函数将设备树目标文件中节点导入到内核，在内核中由device\_node结构体实例的层次结构表示设备节点。

下面先看一下device\_node结构体的定义和实例的组织结构。

device\_node结构体定义在/include/linux/of.h头文件内：

struct device\_node {

const char \*name; /\*设备节点名称\*/

const char \*type; /\*设备节点类型\*/

phandle phandle;

const char \*full\_name;

struct fwnode\_handle fwnode;

struct property \***properties;** /\*属性链表\*/

struct property \*deadprops; /\*删除的属性链表\*/

struct device\_node \*parent; /\*父节点指针\*/

struct device\_node \*child; /\*子节点指针\*/

struct device\_node \*sibling; /\*兄弟节点指针\*/

struct kobject kobj; /\*导出到sysfs文件系统中的kobject实例\*/

unsigned long \_flags;

void \*data;

...

};

设备属性由property结构体表示，定义如下：

struct property {

char \*name; /\*属性名称\*/

int length; /\*属性值长度\*/

void \*value; /\*属性值长度\*/

struct property \*next; /\*下一个属性，构成属性链表\*/

unsigned long \_flags;

unsigned int unique\_id;

struct bin\_attribute attr; /\*属性导出到节点在sysfs文件系统目录下（导出为文件）\*/

};

在体系结构相关的device\_tree\_init()函数中通常调用**unflatten\_and\_copy\_device\_tree()**函数将设备树目标文件导入内核，函数定义如下（/drivers/of/fdt.c）：

void \_\_init unflatten\_and\_copy\_device\_tree(void)

{

int size;

void \*dt;

if (!initial\_boot\_params) { /\*initial\_boot\_params保存设备树地址\*/

...

}

size = fdt\_totalsize(initial\_boot\_params); /\*设备树文件大小\*/

dt = early\_init\_dt\_alloc\_memory\_arch(size,roundup\_pow\_of\_two(FDT\_V17\_SIZE));

/\*为设备树分配空间\*/

if (dt) {

memcpy(dt, initial\_boot\_params, size); /\*复制设备树目标文件\*/

initial\_boot\_params = dt; /\*新复制的设备树目标文件地址\*/

}

**unflatten\_device\_tree();** /\*展开设备树\*/

}

unflatten\_and\_copy\_device\_tree()函数先复制目标文件至内核空间，然后调用unflatten\_device\_tree()函数将设备树目标文件中节点展开成device\_node实例组成的层次结构，如下图所示：



#### 2设备树接口函数

基于device\_node实例的层次结构，内核提供了从device\_node实例中获取设备节点信息的接口函数，这些函数声明在/include/linux/of.h头文件，定义在/drivers/of/base.c文件内，驱动程序中可调用这些接口函数获取设备信息。例如：

●struct device\_node \***of\_find\_node\_by\_name**(struct device\_node \*from,const char \*name)：通过节点名称查找device\_node实例，name表示名称，from表示查找起点，为NULL表示从根节点开始查找。

●struct device\_node \***of\_find\_compatible\_node**(struct device\_node \*from,const char \*type, const char \*compatible)：按照兼容属性查找节点，compatible指向兼容属性字符串。

●struct of\_device\_id \***of\_match\_node**(const struct of\_device\_id \*matches,const struct device\_node \*node)：检查node节点与of\_device\_id列表各项的匹配性，返回匹配的of\_device\_id项指针。

●struct property \***of\_find\_property**(const struct device\_node \*np,const char \*name,int \*lenp)：查找指定节点的指定属性。

●int **of\_property\_read\_string**(struct device\_node \*np, const char \*propname,const char \*\*out\_string)：查找并读取指定节点指定字符串属性值，np指定节点，propname为属性名称，\*out\_string指向属性值字符串。

●int **of\_property\_read\_u32\_array**(const struct device\_node \*np,const char \*propname, u32\*out\_values,size\_t sz)：读取32位属性值（数组），np指定节点，propname为属性名称，\*out\_values指向属性值数组，sz为数组项数（32位整数数量）。

●int **of\_property\_read\_u32\_index**(const struct device\_node \*np,const char \*propname,u32 index, u32 \*out\_value)：读取index指定的32位属性值。

## 8.6平台总线

在了解了驱动模型框架之后，从本节开始介绍几种具体总线（驱动）的实现。

下面先介绍在嵌入式系统中常用的平台（platform）总线。平台总线是内核定义的虚拟总线，用于管理没有挂接到实际总线上（如：SPI、USB总线）的设备，也可以将platform总线理解成处理器的地址/数据总线（芯片内总线）。

### 8.6.1 platform总线

#### 1总线实例

platform总线实例定义在/drivers/base/platform.c文件：

struct bus\_type platform\_bus\_type = {

.name = "platform", /\*总线名称为"platform"\*/

.dev\_groups = platform\_dev\_groups, /\*总线下设备默认属性\*/

.match = platform\_match, /\*设备与驱动匹配函数，见下文\*/

.uevent = platform\_uevent, /\*向uevent事件添加环境变量的函数\*/

.pm = &platform\_dev\_pm\_ops, /\*电源管理操作\*/

}; /\*总线没有定义探测函数\*/

platform\_bus\_type实例中部分成员简介如下：

**●platform\_dev\_groups：**默认设备属性指针数组，定义在/drivers/base/platform.c文件内，表示默认添加到platform总线下设备的属性。

static struct attribute \*platform\_dev\_attrs[] = {

&dev\_attr\_modalias.attr, /\*modalias只读属性，获取模块别名\*/

&dev\_attr\_driver\_override.attr, /\*driver\_override读写属性，属性值为**pdev->driver\_override**\*/

NULL,

};

总线设备默认属性包括modalias和driver\_override，它们都定义在/drivers/base/platform.c文件内，属性读写函数请读者自行阅读。

**●platform\_dev\_pm\_ops：**电源管理数据结构，实例定义在/drivers/base/platform.c文件内：

static const struct dev\_pm\_ops platform\_dev\_pm\_ops = {

.runtime\_suspend = pm\_generic\_runtime\_suspend, /\*/drivers/base/power/generic\_ops.c\*/

.runtime\_resume = pm\_generic\_runtime\_resume,

USE\_PLATFORM\_PM\_SLEEP\_OPS

};

platform\_dev\_pm\_ops实例中各函数，在没有选择PM配置选项时直接返回0，否则各函数将调用设备驱动device\_driver->pm指向实例中定义对应函数，如果驱动中没有定义对应的函数则返回0。

**●platform\_match()：**判断总线下指定设备与指定驱动是否匹配的函数，若匹配函数返回1，否则返回0，函数实现后面再做介绍。

●**platform\_uevent()**：向总线添加设备触发uevent事件时，调用此函数添加环境变量。此函数定义在/drivers/base/platform.c文件内，代码如下：

static int platform\_uevent(struct device \*dev, struct kobj\_uevent\_env \*env)

{

struct platform\_device \*pdev = to\_platform\_device(dev);

int rc;

rc = of\_device\_uevent\_modalias(dev, env); /\*添加模块别名的环境变量\*/

if (rc != -ENODEV)

return rc;

rc = acpi\_device\_uevent\_modalias(dev, env);

if (rc != -ENODEV)

return rc;

add\_uevent\_var(env, "MODALIAS=%s%s", PLATFORM\_MODULE\_PREFIX,pdev->name);

return 0;

}

#### 2初始化

platform总线初始化函数platform\_bus\_init(void)完成总线的注册，函数由driver\_init()函数调用，初始化函数定义在/drivers/base/platform.c文件内，代码如下：

int \_\_init platform\_bus\_init(void)

{

int error;

early\_platform\_cleanup();

error = **device\_register(&platform\_bus)**; /\*将总线当成设备向内核注册，名称为"platform"\*/

if (error)

return error;

error = **bus\_register(&platform\_bus\_type)**; /\*注册总线实例\*/

if (error)

device\_unregister(&platform\_bus);

**of\_platform\_register\_reconfig\_notifier();** /\*注册通知\*/

return error;

}

初始化函数比较简单，platform总线被当成设备（或视其为主机控制器）注册为/sys/devices/*platform*目录，作为总线下设备的父设备，然后注册总线**platform\_bus\_type**实例。

### 8.6.2 platform设备

platform\_device结构体表示挂接在platform总线上的设备，它是device结构体的包装器。

#### 1数据结构

内核在/include/linux/platform\_device.h头文件内定义了platform\_device结构体，表示平台总线设备：

struct platform\_device {

const char \***name**; /\*设备名称\*/

int **id**; /\*设备编号，由内核管理设备编号，platform\_devid\_ida\*/

bool id\_auto;

struct device **dev**; /\*内嵌device实例\*/

u32 num\_resources; /\*资源实例数量\*/

struct resource **\*resource**; /\*指向资源结构实例数组\*/

const struct platform\_device\_id **\*id\_entry**; /\*指向驱动platform\_device\_id列表中的匹配项\*/

char **\*driver\_override**; /\*强制匹配的驱动名称，只进行与驱动名称的匹配\*/

struct mfd\_cell \*mfd\_cell;

struct pdev\_archdata archdata; /\*体系结构相关的数据结构，MIPS为空\*/

};

platform\_device结构体中主要成员简介如下：

**●name**：设备名称，在总线匹配函数中用于匹配驱动，在添加设备时将赋予device实例成员（或尾部加上编号）。

●**id**：设备在平台总线上的编号，设为**PLATFORM\_DEVID\_NONE**(-1)表示向总线添加设备时不分配编号，为PLATFORM\_DEVID\_AUTO(-2)表示由内核自动分配编号，并赋予id成员。在注册设备时name加上编号id将赋予dev实例作为设备在sysfs文件系统中的名称。

**●dev：**内嵌device结构体实例。

●**resource：**资源结构resource实例指针，实例指向的是资源数组，后面再作介绍。

●**driver\_override：**字符串指针，如果设置了此成员，则在匹配驱动时，只用此字符串与驱动名称进行匹配，不再进行其它方式的匹配。

**●id\_entry：**指向匹配的platform\_device\_id实例。在platform驱动中如果定义了platform\_device\_id实例数组，则在匹配设备与驱动的函数中，会将设备名称与驱动中platform\_device\_id实例数组项（name成员）进行比较，名称相同则表示匹配上。id\_entry成员指向与本设备匹配的数组项（platform\_device\_id实例），详见下文件platform驱动中的介绍。

#### 2资源

资源由resource结构体表示，是设备具有的硬件资源，如内存、中断号等。

##### **■数据结构**

资源resource结构体定义在/include/linux/ioport.h头文件：

struct resource {

resource\_size\_t start; /\*起始值\*/

resource\_size\_t end; /\*结束值\*/

const char \*name; /\*资源名称\*/

unsigned long **flags**; /\*标记\*/

struct resource \*parent, \*sibling, \*child; /\*资源结构实例组成父子层次结构\*/

};

resource结构体是描述各种硬件资源的通用数据结构，主要成员简介如下：

●start、end：通用的资源的起始和结束值，如内存的起始、结束地址。

●name：资源名称。

●parent、sibling、child：resource实例指针，用于将实例组织成父子层次结构。父子资源之间是包含关系，即父资源的起止值包含了其下子资源的起止值。

●flags：标记。flags成员为32位无符号整数，各标记位语义定义在/include/linux/ioport.h头文件。其中bit[7..0]共8位表示特定于总线的特性，bit[12..8]共5位表示资源的类型，其它标记位表示属性的其它特性，如下所示。

#define IORESOURCE\_BITS 0x000000ff /\*总线特性掩码，共8位\*/

#define IORESOURCE\_TYPE\_BITS 0x00001f00 /\*资源类型掩码，共5位\*/

**#define IORESOURCE\_IO 0x00000100** /\* PCI/ISA I/O ports，IO端口\*/

**#define IORESOURCE\_MEM 0x00000200**  /\*内存资源\*/

#define IORESOURCE\_REG 0x00000300 /\* IO寄存器偏移量\*/

**#define IORESOURCE\_IRQ 0x00000400** /\*中断编号\*/

#define IORESOURCE\_DMA 0x00000800

#define IORESOURCE\_BUS 0x00001000

#define IORESOURCE\_PREFETCH 0x00002000 /\*没有副作用\*/

#define IORESOURCE\_READONLY 0x00004000 /\*只读资源\*/

#define IORESOURCE\_CACHEABLE 0x00008000 /\*可缓存\*/

#define IORESOURCE\_RANGELENGTH 0x00010000

#define IORESOURCE\_SHADOWABLE 0x00020000

#define IORESOURCE\_SIZEALIGN 0x00040000 /\*大小（长度）值对齐\*/

#define IORESOURCE\_STARTALIGN 0x00080000 /\*起始值对齐\*/

#define IORESOURCE\_MEM\_64 0x00100000

#define IORESOURCE\_WINDOW 0x00200000 /\* forwarded by bridge \*/

#define IORESOURCE\_MUXED 0x00400000 /\* Resource is software muxed \*/

#define IORESOURCE\_EXCLUSIVE 0x08000000 /\*独占式资源\*/

#define IORESOURCE\_DISABLED 0x10000000

#define IORESOURCE\_UNSET 0x20000000 /\* No address assigned yet \*/

#define IORESOURCE\_AUTO 0x40000000

#define IORESOURCE\_BUSY 0x80000000 /\*驱动程序已标记资源忙\*/

/\* 以下是PnP 各类型资源的特性标记，低8位\*/

... （略）

/\* PCI ROM控制器资源特性位\*/

#define IORESOURCE\_ROM\_ENABLE (1<<0)

#define IORESOURCE\_ROM\_SHADOW (1<<1)

#define IORESOURCE\_ROM\_COPY (1<<2)

#define IORESOURCE\_ROM\_BIOS\_COPY (1<<3)

/\* PCI控制位 \*/

#define IORESOURCE\_PCI\_FIXED (1<<4) /\* Do not move resource \*/

##### **■初始化**

资源管理是为了更好地分配资源，防止冲突，设备需要使用资源时需先向内核申请。资源管理相关代码位于/kernel/resource.c文件内，先来看初始化函数。

内核在/kernel/resource.c文件内定义了IO端口和物理内存两个资源类型的根节点：

struct resource ioport\_resource = {

.name = "PCI IO",

.start = 0,

.end = IO\_SPACE\_LIMIT,

.flags = IORESOURCE\_IO, /\*类型，IO资源\*/

};

IO\_SPACE\_LIMIT表示端口结束地址，它显然应该是一个与体系结构相关的参数。MIPS体系结构下，定义在以/arch/mips/include/asm/io.h头文件内：

#define IO\_SPACE\_LIMIT 0xffff

物理内存资源根节点定义如下：

struct resource iomem\_resource = {

.name = "PCI mem",

.start = 0,

.end = -1,

.flags = IORESOURCE\_MEM, /\*类型，物理内存资源\*/

};

内核在启动初期需要对资源管理数据结构进行初始化，资源管理结构其实就是由resource实例组成的层次树状结构，每种类型资源由一个层次结构管理。父子资源包含子资源。

资源初始化函数调用关系为setup\_arch()->resource\_init()，resource\_init()函数在/arch/mips/kernel/setup.c文件内实现，代码如下：

static struct resource code\_resource = { .name = "Kernel code", }; /\*内核代码所占内存\*/

static struct resource data\_resource = { .name = "Kernel data", }; /\*内核数据所占内存\*/

static void \_\_init resource\_init(void)

{

int i;

if (UNCAC\_BASE != IO\_BASE)

return;

code\_resource.start = \_\_pa\_symbol(&\_text); /\*设置内核代码、数据内存的起止地址\*/

code\_resource.end = \_\_pa\_symbol(&\_etext) - 1;

data\_resource.start = \_\_pa\_symbol(&\_etext);

data\_resource.end = \_\_pa\_symbol(&\_edata) - 1;

for (i = 0; i < boot\_mem\_map.nr\_map; i++) { /\*扫描物理内存段信息，注册内存资源\*/

struct resource \*res;

unsigned long start, end;

start = boot\_mem\_map.map[i].addr;

end = boot\_mem\_map.map[i].addr + boot\_mem\_map.map[i].size - 1;

if (start >= HIGHMEM\_START)

continue;

if (end >= HIGHMEM\_START)

end = HIGHMEM\_START - 1; /\*剔除高端内存\*/

res = **alloc\_bootmem(sizeof(struct resource))**; /\*从自举分配器中分配资源结构实例\*/

switch (boot\_mem\_map.map[i].type) {

case BOOT\_MEM\_RAM: /\*设置资源名称\*/

case BOOT\_MEM\_ROM\_DATA:

res->name = "System RAM";

break;

case BOOT\_MEM\_RESERVED:

default:

res->name = "reserved";

}

res->start = start;

res->end = end;

res->flags = **IORESOURCE\_MEM | IORESOURCE\_BUSY**; /\*资源实例初始化\*/

**request\_resource(&iomem\_resource, res)**; /\*在iomem\_resource下注册新资源实例\*/

**request\_resource(res, &code\_resource)**; /\*code\_resource注册到新资源实例下\*/

**request\_resource(res, &data\_resource)**; /\*data\_resource注册到新资源实例下\*/

}

}

resource\_init()函数扫描boot\_mem\_map实例保存的物理内存段信息，对每个内存段创建resource实例（内存类型），然后将resource实例注册到**iomem\_resource**物理内存资源实例下。新创建的resource实例名称为"System RAM"或"reserved"，最后尝试将code\_resource和data\_resource实例注册到新创建的resource实例下。

resource\_init()函数执行后资源数据结构实例组织关系如下图所示：



在/kernel/resource.c文件还定义了ioresources\_init()函数，在proc文件系统下创建两个文件，如下所示：

static int \_\_init ioresources\_init(void)

{

proc\_create("ioports", 0, NULL, &proc\_ioports\_operations); /\*ioports文件\*/

proc\_create("iomem", 0, NULL, &proc\_iomem\_operations); /\*iomem文件\*/

return 0;

}

\_\_initcall(ioresources\_init);

ioports文件和iomem文件是只读文件，用于输出IO和内存资源信息。

##### **■申请资源**

在前面介绍的初始化中，内核定义了IO和内存资源的根节点ioport\_resource和iomem\_resource实例，所有IO及内存资源都应位于此根节点之下，并且不能超过根节点对资源的限制值。

在根节点下申请（注册）资源的函数为request\_resource(struct resource \*root, struct resource \*new)，函数定义在/kernel/resource.c文件内，代码如下：

int request\_resource(struct resource \*root, struct resource \*new)

/\*root：资源根节点，new：申请的新资源实例\*/

{

struct resource \*conflict;

conflict = **request\_resource\_conflict(root, new)**; /\*/kernel/resource.c\*/

return conflict ? -EBUSY : 0; /\*申请成功返回0，否则返回错误码-EBUSY\*/

}

root为new资源实例的根节点，new表示申请的资源。new表示资源的起止值必须在root表示资源的起止值范围内，也就是说root必须包含new，且new不与root下已有所有子资源值范围有重叠。资源申请成功返回0，不成功返回错误码。

request\_resource()函数直接调用request\_resource\_conflict(root, new)函数完成具体的工作，此函数获取锁后调用\_\_request\_resource(root, new)函数完成资源的申请，随后释放锁。

\_\_request\_resource(root, new)函数定义如下：

static struct resource \* \_\_request\_resource(struct resource \*root, struct resource \*new)

{

resource\_size\_t start = new->start; /\*新资源起始值\*/

resource\_size\_t end = new->end; /\*结束值\*/

struct resource \*tmp, \*\*p;

if (end < start)

return root;

if (start < root->start)

return root;

if (end > root->end)

return root; /\*判断资源信息是否有效，无效返回root\*/

**p = &root->child**; /\*指向root的子节点\*/

for (;;) { /\*遍历子节点，找到new合适的插入点，并插入父节点的子节点链表\*/

tmp = \*p;

if (!tmp || tmp->start > end) { /\*没有重叠\*/

**new->sibling = tmp;**  /\*插入子链表\*/

**\*p = new;**

new->parent = root;

return NULL; /\*申请成功（没有检查重叠）\*/

}

p = &tmp->sibling;

if (tmp->end < start) /\*与当前子节点没有重叠，遍历下一节点\*/

continue;

return tmp; /\*返回tmp表示new与tmp有重叠，申请失败\*/

}

}

\_\_request\_resource()函数只遍历root下的子资源实例（不遍历子节点的子节点），查找new实例合适的插入点。子资源实例在链表中以起始值从小到大在链表中从左至右排列。如果new与现有root下子资源实例值范围有重叠，函数将返回有重叠资源实例的指针，申请失败。如果没有重叠，则将new实例插入正root下的子资源链表，函数返回0，申请成功。

在指定父节点下申请（注册）资源的函数为int **insert\_resource(struct resource \*parent, struct resource \*new)**，parent为父节点，new为申请（注册）节点，此函数会逐层遍历parent下的子孙节点，如果new与现在节点没有重叠则直接添加，如果new包含于某个节点内部，则添加为此节点的子节点，如果与某个节点部分重叠则返回错误码，函数源代码请读者自行阅读。

int **devm\_request\_resource**(struct device \*dev, struct resource \*root,struct resource \*new)：用于将resource实例作为device实例的资源向device实例添加，devres实例后面接的是new指向resource实例的副本，如下图所示。new指向实例添加到管理结构中，添加到root下子节点链表中（调用request\_resource\_conflict()函数），源代码请读者自行阅读。



int **platform\_device\_add\_resources**(struct platform\_device \*pdev,const struct resource \*res, unsigned int num)：用于将platform\_device实例关联资源数组，res指向资源数组，num为数组项数。

#### 3注册设备

在体系结构相关的代码（板级相关的代码）可静态定义platform\_device实例，在内核启动阶段，向platform总线注册设备。

如果使用了设备树，将根据设备节点device\_node实例动态创建和注册platform\_device实例。

##### ■静态注册

如果没有使用设备树，注册platform设备的函数为platform\_device\_register(pdev)，定义在/drivers/base/platform.c文件内：

int **platform\_device\_register**(struct platform\_device \*pdev)

{

**device\_initialize**(&pdev->dev); /\*初始化内嵌device实例\*/

arch\_setup\_pdev\_archdata(pdev); /\*空操作\*/

return **platform\_device\_add(pdev)**; /\*添加platform\_device设备\*/

}

device\_initialize(&pdev->dev)函数在前面介绍过，用于初始化platform\_device内嵌的device实例。

platform\_device\_add(pdev)函数完成向内核添加platform\_device设备的工作，代码如下：

int platform\_device\_add(struct platform\_device \*pdev)

{

int i, ret;

if (!pdev)

return -EINVAL;

if (!pdev->dev.parent)

**pdev->dev.parent = &platform\_bus;**  /\*父设备为表示platform\_bus总线的设备\*/

**pdev->dev.bus = &platform\_bus\_type**; /\*platform\_bus总线\*/

**switch (pdev->id)** { /\*如果设置了id成员，需要设置pdev->dev名称\*/

default: /\*静态指定id编号\*/

dev\_set\_name(&pdev->dev, "%s.%d", pdev->name, pdev->id); /\*device名称为name+id\*/

break;

case **PLATFORM\_DEVID\_NONE**: /\*不使用id编号（-1）\*/

dev\_set\_name(&pdev->dev, "%s", pdev->name); /\*直接使用pdev->name作为设备名称\*/

break;

case PLATFORM\_DEVID\_AUTO: /\*由内核自动分配id编号\*/

ret = ida\_simple\_get(&platform\_devid\_ida, 0, 0, GFP\_KERNEL);

if (ret < 0)

goto err\_out;

pdev->id = ret;

pdev->id\_auto = true;

dev\_set\_name(&pdev->dev, "%s.%d.auto", pdev->name, pdev->id); /\*设置名称\*/

break;

}

for (i = 0; i < pdev->num\_resources; i++) { /\*添加资源，扫描资源数组\*/

struct resource \*p, \*r = &pdev->resource[i];

if (r->name == NULL)

r->name = dev\_name(&pdev->dev); /\*设置资源名称\*/

p = r->parent;

if (!p) { /\*如果资源没有指定父节点，则根据资源类型设置父节点\*/

if (resource\_type(r) == IORESOURCE\_MEM)

p = &**iomem\_resource**;

else if (resource\_type(r) == IORESOURCE\_IO)

p = &**ioport\_resource**;

}

if (p && **insert\_resource(p, r)**) { /\*存在父节点，并执行插入资源操作\*/

...

}

}

...

**ret = device\_add(&pdev->dev);** /\*向内核添加设备device实例\*/

if (ret == 0)

return ret;

...

}

platform\_device\_add()函数主要完成内嵌device实例父节点的设置、设置所属总线、设置名称等，向内核注册设备资源，最后调用device\_add()向内核添加内嵌的device实例。

void platform\_device\_unregister(struct platform\_device \*pdev)：注销platform\_device实例。

int **platform\_add\_devices**(struct platform\_device \*\*devs, int num)：注册platform\_device实例数组，参数devs指向platform\_device实例的指针数组，num为需要添加的实例数量，调用platform\_device\_register()函数执行注册设备操作。

int **platform\_device\_add\_data**(struct platform\_device \*pdev, const void \*data,size\_t size)：将设备的平台信息data指向的数据结构赋予pdev->dev.platform\_data成员，size为数据大小。

##### ■动态注册

如果使用了设备树，用户通过设备树文件向内核传递设备硬件信息。外部设备在设备树中由节点表示，在内核中由device\_node结构体实例表示。

内核在/drivers/of/platform.c文件内定义了为指定设备节点创建platfrom\_device实例的函数：

struct platform\_device \***of\_platform\_device\_create**(struct **device\_node \*np**,const char \*bus\_id,

struct device \*parent)；

/\*调用platform\_device\_alloc()函数分配platform\_device实例\*/

struct platform\_device \***platform\_device\_alloc**(const char \*name, int id)：用于动态分配platform\_device实例（/drivers/base/platform.c）。

struct platform\_device \***platform\_device\_register\_full**(const struct platform\_device\_info \*pdevinfo)：创建并注册带资源和平台数据的platform\_device实例。

platform\_device\_info结构体定义如下（/include/linux/platform\_device.h）：

struct platform\_device\_info {

struct device \***parent**; /\*父设备\*/

struct fwnode\_handle \*fwnode;

const char \*name; /\*名称\*/

int id;

const struct resource \***res**; /\*资源列表\*/

unsigned int num\_res; /\*资源数量\*/

const void **\*data**; /\*平台数据\*/

size\_t size\_data; /\*平台数据大小\*/

u64 dma\_mask;

};

在后面介绍具体设备驱动程序时，还将涉及到platform\_device的操作函数。

如果平台通过设备树传递platform总线上的设备信息，在平台定义的初始化函数中将调用函数：

int **of\_platform\_bus\_probe**(struct device\_node \*root,const struct of\_device\_id \*matches,

struct device \*parent); /\*/drivers/of/platform.c\*/

扫描设备树节点，检查节点与matches指定of\_device\_id实例是否匹配，若匹配则创建并添加表示设备的platform\_device实例，将从节点中获取设备信息填充至platform\_device实例。

其它总线在注册主机控制器时，将扫描设备树中控制器下的设备节点，为其创建并设置xxx\_device实例。

### 8.6.3 platform驱动

平台总线驱动由platform\_driver结构体表示，它是通用驱动模型中device\_driver结构体的包装器。

#### 1 数据结构

platform总线设备驱动platform\_driver结构体定义如下（/include/linux/platform\_device.h）：

struct platform\_driver {

int (\***probe**) (struct platform\_device \*); /\*探测设备函数，匹配设备后调用\*/

int (\*remove) (struct platform\_device \*); /\*移除platform设备时调用\*/

void (\*shutdown) (struct platform\_device \*); /\*关闭设备时调用\*/

int (\*suspend) (struct platform\_device \*, pm\_message\_t state); /\*设备睡眠时调用\*/

int (\*resume) (struct platform\_device \*); /\*唤醒设备时调用\*/

struct device\_driver  **driver;**  /\*内嵌device\_driver结构体实例\*/

const struct platform\_device\_id  **\*id\_table;**  /\*匹配列表\*/

bool prevent\_deferred\_probe;

};

platform\_driver结构体主要成员简介如下：

●**probe()**：探测函数，设备与驱动匹配后调用此函数，加载设备驱动程序。

●**id\_table：**指向platform\_device\_id实例数组，每个数组项保存匹配本驱动的设备名称，用于与platform\_device实例进行名称匹配，列表最后一项必须为空。

platform\_device\_id结构体定义在/include/linux/mod\_devicetable.h头文件：

#define PLATFORM\_NAME\_SIZE 20

#define PLATFORM\_MODULE\_PREFIX "platform:"

struct platform\_device\_id {

char name[PLATFORM\_NAME\_SIZE]; /\*匹配字符串，用于与设备名称匹配\*/

kernel\_ulong\_t driver\_data;

};

#### 2 注册驱动

platform\_driver实例通常在设备驱动程序中静态定义，在内核初始化函数或模块初始化函数中调用注册函数**platform\_driver\_register(drv)**向内核注册实例。

注册函数定义在/include/linux/platform\_device.h头文件内：

#define platform\_driver\_register(drv) \

\_\_platform\_driver\_register(drv, THIS\_MODULE)

\_\_platform\_driver\_register()函数在/drivers/base/platform.c文件内实现，代码如下：

int \_\_platform\_driver\_register(struct platform\_driver \*drv,struct module \*owner)

{

drv->driver.owner = owner;

drv->driver.bus = &platform\_bus\_type; /\*指向总线实例\*/

**if (drv->probe)**  /\*如果platform\_driver定义了probe()函数\*/

**drv->driver.probe = platform\_drv\_probe;**  /\*device\_driver.probe赋值通用探测函数\*/

if (drv->remove)

**drv->driver.remove = platform\_drv\_remove;** /\*device\_driver.remove赋值通用函数\*/

if (drv->shutdown)

**drv->driver.shutdown = platform\_drv\_shutdown;** /\*device\_driver.shtdown赋值通用函数\*/

return **driver\_register(&drv->driver)**; /\*注册驱动device\_driver实例\*/

}

注册函数比较简单，需要注意的是platform\_driver结构体中定义了与device\_driver结构体中同名的函数，通常实现platform\_driver实例中的函数，而device\_driver实例中相应的函数赋值为通用函数。通用函数内部调用platform\_driver实例中的同名的函数完成操作，因为通用驱动模型中调用的是device\_driver实例中函数。

如下图所示，假设platform\_driver实例定义了probe()函数，则内嵌device\_driver实例的probe()函数赋值为通用函数platform\_drv\_probe()。在通用驱动模型中，当设备与驱动匹配时，将调用device\_driver实例的probe()函数（platform总线没有定义probe()函数），进而调用platform\_driver实例中的probe()函数，加载设备驱动程序。



platform\_drv\_probe()函数定义如下：

static int platform\_drv\_probe(struct device \*\_dev)

/\*\_dev：platform\_driver结构体内嵌的device实例指针\*/

{

struct platform\_driver \*drv = to\_platform\_driver(\_dev->driver);

struct platform\_device \*dev = to\_platform\_device(\_dev);

int ret;

ret = of\_clk\_set\_defaults(\_dev->of\_node, false);

if (ret < 0)

return ret;

ret = dev\_pm\_domain\_attach(\_dev, true);

if (ret != -EPROBE\_DEFER) {

**ret = drv->probe(dev)**; /\*调用platform\_driver实例probe()函数\*/

if (ret)

dev\_pm\_domain\_detach(\_dev, true);

}

if (drv->prevent\_deferred\_probe && ret == -EPROBE\_DEFER) {

...

}

return ret;

}

platform\_drv\_remove()和platform\_drv\_shutdown()函数与platform\_drv\_probe()函数类似，请读者自行阅读源代码。

void platform\_driver\_unregister(struct platform\_driver \*drv)：用于注销platform驱动。

在/include/linux/platform\_device.h头文件和/drivers/base/platform.c文件内还定义了其它platform\_driver实例的操作函数，例如：

●int \_\_init\_or\_module **\_\_platform\_driver\_probe**(struct platform\_driver \*drv,int (\***probe**)(struct platform\_device \*), struct module \*module)：为不支持热插拔的设备注册驱动，只在注册驱动时会匹配设备，之后设备不能再与本驱动匹配了（probe()函数设为NULL），probe()为探测函数指针。

●**platform\_driver\_probe**(drv, probe)：\_\_platform\_driver\_probe()函数的包装器，module参数为THIS\_MODULE。

●void platform\_set\_drvdata(struct platform\_device \*pdev,void \*data)：用data设置pdev->dev->driver\_data成员，表示驱动数据。

●void \*platform\_get\_drvdata(const struct platform\_device \*pdev)：获取pdev->dev->driver\_data值。

●**module\_platform\_driver**(\_\_platform\_driver)：宏定义，\_\_platform\_driver为platform\_driver实例，定义模块初始化函数为platform\_driver\_register()，表示在加载模块时注册platform\_driver实例，模块卸载函数为platform\_driver\_unregister()，用于卸载模块时注销platform\_driver实例。

●**module\_platform\_driver\_probe**(\_\_platform\_driver, **\_\_platform\_probe**)：宏定义，\_\_platform\_driver为platform\_driver实例，定义模块初始化函数为platform\_driver\_probe()，表示在加载模块时注册platform\_driver实例，其pobe()函数为\_\_platform\_probe()，模块卸载函数为platform\_driver\_unregister()，用于卸载模块时注销platform\_driver实例。

●**builtin\_platform\_driver\_probe**(\_\_platform\_driver, \_\_platform\_probe)：宏定义，表示在内核初始化时调用platform\_driver\_probe()函数，注册platform\_driver实例，其pobe()函数为\_\_platform\_probe()，此宏不适用于模块。

### 8.6.4总线匹配函数

在向总线添加设备/驱动时，将扫描总线的驱动/设备链表，对各实例调用总线定义的match()函数判断设备与驱动是否匹配，如果匹配则调用总线或驱动的probe()函数，加载设备驱动程序。

下面看一下platform总线匹配函数的定义（/drivers/base/platform.c）：

static int platform\_match(struct device \*dev, struct device\_driver \*drv)

{

struct platform\_device \*pdev = to\_platform\_device(dev);

struct platform\_driver \*pdrv = to\_platform\_driver(drv);

if (**pdev->driver\_override**) /\*如果设置了driver\_override，则只将其与驱动名称进行匹配\*/

return !strcmp(pdev->driver\_override, drv->name); /\*strcmp()字符相等返回0\*/

if (**of\_driver\_match\_device(dev, drv)**) /\*匹配设备树节点，/include/linux/of\_device.h\*/

return 1;

if (acpi\_driver\_match\_device(dev, drv)) /\*需选择ACPI配置选项，/drivers/acpi/scan.c\*/

return 1;

if (pdrv->id\_table) /\*platform\_device\_id实例列表匹配\*/

return **platform\_match\_id(pdrv->id\_table, pdev) != NULL**;

return (strcmp(pdev->name, drv->name) == 0); /\*最后进行设备与驱动名称的匹配\*/

}

platform\_match()函数判断设备与驱动是否匹配的顺序如下（有一项匹配则返回1）：

（1）如果platform\_device设置了**driver\_override**成员则只将其与驱动名称进行匹配，匹配返回1，不匹配返回0，不再进行下面的匹配。下面步骤以driver\_override成员为NULL为前提。

（2）通过device\_driver->driver->of\_match\_table列表检查与platform\_device->dev->of\_node是否匹配，比较兼容属性。

（3）ACPI列表匹配。

（4）将platform\_device实例与platform\_driver实例中platform\_device\_id实例数组项匹配，见下文。

（5）如果以上匹配都不成功，最后检查设备与驱动名称的是否相同，相同则表示匹配，返回1，不相同表示不匹配，返回0。

下面看一下检查platform\_device实例与platform\_driver实例中platform\_device\_id实例数组项匹配的函数。platform\_match\_id()函数定义如下（/drivers/base/platform.c）：

static const struct platform\_device\_id \*platform\_match\_id(const struct platform\_device\_id \*id, \

struct platform\_device \*pdev)

/\*id：驱动platform\_device\_id实例数组指针，pdev：platform\_device指针\*/

{

while (id->name[0]) { /\*直至扫描到空的数组项\*/

if (**strcmp(pdev->name, id->name**) == 0) { /\*将设备名称与列表项name成员比较\*/

**pdev->id\_entry = id**; /\*相同则**pdev->id\_entry**指向匹配的platform\_device\_id实例\*/

return id;

}

id++;

}

return NULL;

}

platform\_match\_id()函数将platform\_device名称依次与驱动中的platform\_device\_id实例数组项的名称进行比较，若相同则表示匹配成功，并将匹配项赋予platform\_device实例的id\_entry成员，函数返回匹配列表项指针，否则返回NULL。

## 8.7 SPI总线

SOC系统中，SPI、I2C、USB等总线结构如下图所示，处理器（主机）具有总线的主机控制器，负责通过总线连接外接设备，控制器以特定的协议在总线上产生波形，实现与外接设备的数据传输和进行设备控制。主机控制器挂载在芯片内部总线上，通常处理器通过操作控制器的寄存器来对控制器进行控制和数据传输，处理器对主机控制器的操作类似于对物理内存的操作。



Linux内核采用主机控制器驱动与外设驱动分离的思想。总线控制驱动只负责产生波形，传输给定的数据，数据是何用处，数据的内容是什么，主机控制器不管。主机控制器驱动向外设驱动程序提供传输数据的接口函数，数据如何解释和使用由外设驱动程序实现。

外设驱动程序调用主机控制器驱动提供的接口函数在总线上实现数据的传输。外设驱动只需要提供传输数据的内容，调用接口函数进行数据传输即可，它并不需要关心数据是如何进行传输的，这由主机控制器驱动实现。

本节介绍SPI主机控制器（总线）驱动的实现，后面将介绍I2C、USB等总线（控制器）驱动的实现。

### 8.7.1概述

SPI总线是一种全双工串行通信总线，接口协议中包括时钟信号、片选信号、主机发送信号线及主机接收信号线等。SPI总线结构如下图所示，处理器通过操作SPI控制器的寄存器来实现总线上数据的传输。SPI总线上可能挂接了多个设备，它们共用总线进行数据传输，通过控制器的片选信号来确定SPI控制器与哪个外接设备进行数据传输。



SPI总线驱动框架如下图所示：



图中上半部分表示SPI主机控制器驱动，下半部分表示SPI设备驱动。SPI主机控制器在驱动框架中由**spi\_master**结构体表示，结构体中包含控制器的物理信息及数据传输函数指针等成员，结构体实例在内核中由全局双链表管理。主机控制器作为一个设备挂接在平台总线上（或其它总线，下同），在添加表示控制器的设备时（platform\_device），匹配控制器驱动（platform\_driver），匹配成功调用驱动中的probe()函数，函数内完成spi\_master实例的创建和注册。

SPI主机控制器的spi\_master实例中具有需要传输数据的消息队列，spi\_master结构体中函数指针成员transfer\_one\_message()用于完成一个消息的数据传输。SPI控制器驱动程序的主要工作就是在驱动的probe()函数中创建和设置spi\_master实例，并向内核注册。

SPI驱动框架中定义了spi\_bus\_type总线，SPI总线设备和驱动挂载到此总线上。SPI设备由spi\_device结构体表示，SPI设备驱动由spi\_driver结构体表示。向SPI总线添加设备或驱动时，将会在总线下查找匹配的驱动或设备，匹配成功将调用驱动的probe()函数，函数内完成具体字符/块设备驱动程序的加载（如：注册cdev实例）。

SPI驱动框架中提供了向spi\_master实例提交消息，执行数据传输的接口函数，SPI设备驱动程序中文件操作结构file\_operations实例中的函数调用此接口函数进行数据传输。

向SPI总线添加设备的操作与平台设备的添加稍有不同，用户需通过**spi\_register\_board\_info()**接口函数注册**spi\_board\_info**实例来完成SPI设备的添加。spi\_board\_info结构体表示SPI设备的信息，注册此结构体实例时会查找匹配的spi\_master实例（设备连接到的SPI主机控制器），若匹配成功则创建**spi\_device**结构体实例，并挂载到SPI总线。在板级相关的文件内只需要为SPI设备创建和注册spi\_board\_info实例即可，而不需要创建和添加spi\_device实例。

### 8.7.2控制器驱动

SPI主机控制器驱动主要完成spi\_master实例的创建和注册，以及数据传输函数的实现，控制器驱动位于/drivers/spi/目录下。

#### 1数据结构

每个SPI主机控制器由spi\_master结构体表示，结构体定义如下（/include/linux/spi/spi.h）：

struct spi\_master {

struct device  **dev**; /\*内嵌device实例，添加到通用驱动模型中\*/

struct list\_head **list**; /\*双链表成员，将spi\_master 实例添加到全局链表\*/

s16  **bus\_num;** /\*主机控制器编号，从0开始编号，用于标识SPI主机控制器\*/

u16 **num\_chipselect**; /\*片选信号最大编号，从0开始编号，用于区分外接设备\*/

u16 dma\_alignment; /\*DMA对齐要求\*/

u16 mode\_bits; /\*识别spi\_device.mode成员的位数\*/

u32 bits\_per\_word\_mask; /\* bitmask of supported bits\_per\_word for transfers \*/

...

u32 min\_speed\_hz; /\*最低传输速率\*/

u32 max\_speed\_hz; /\*最高传输速率\*/

u16 flags; /\*标记成员，取值定义如下\*/

...

spinlock\_t bus\_lock\_spinlock; /\*保护自旋锁\*/

struct mutex bus\_lock\_mutex; /\*保护互斥量\*/

bool bus\_lock\_flag; /\*标记总线被独占使用\*/

int (\***setup**)(struct spi\_device \*spi); /\*修改传输速率和模式\*/

int (\*transfer)(struct spi\_device \*spi,struct spi\_message \*mesg); /\*已弃用\*/

/\*进行消息传输的函数，数据由spi\_message结构体表示\*/

void (\*cleanup)(struct spi\_device \*spi); /\*释放由spi\_master提供的内存\*/

bool (\*can\_dma)(struct spi\_master \*master, struct spi\_device \*spi,struct spi\_transfer \*xfer);

/\*是否支持DMA\*/

bool queued; /\*是否提供内部消息队列\*/

struct kthread\_worker **kworker;**  /\*内核线程工作者\*/

**s**truct task\_struct  **\*kworker\_task**; /\*内核工作者进程结构指针\*/

struct kthread\_work **pump\_messages**; /\*内核工作者工作\*/

spinlock\_t queue\_lock; /\*消息队列保护自旋锁\*/

struct list\_head **queue**; /\*消息队列，管理spi\_messages实例\*/

struct spi\_message \*cur\_msg; /\*当前正在处理的消息\*/

bool idling; /\*设备进入空闲状态\*/

bool busy; /\*忙状态，正在处理消息\*/

bool running; /\*正在处理消息\*/

bool  **rt;** /\*处理消息队列的线程是否设为实时线程\*/

bool auto\_runtime\_pm;

bool cur\_msg\_prepared; /\*是否调用spi\_prepare\_message\*/

bool cur\_msg\_mapped; /\*消息是否已映射到DMA\*/

**struct completion xfer\_completion**; /\*完成量，由transfer\_one\_message()使用\*/

size\_t max\_dma\_len; /\*DMA最大传输长度\*/

int (\*prepare\_transfer\_hardware)(struct spi\_master \*master);

int (\***transfer\_one\_message**)(struct spi\_master \*master,struct spi\_message \*mesg);

/\*传输单个spi\_message的函数，数据格式为**spi\_message**\*/

int (\*unprepare\_transfer\_hardware)(struct spi\_master \*master);

int (\*prepare\_message)(struct spi\_master \*master,struct spi\_message \*message);

int (\*unprepare\_message)(struct spi\_master \*master,struct spi\_message \*message);

void (\***set\_cs**)(struct spi\_device \*spi, bool enable); /\*设置片选信号\*/

int (\***transfer\_one**)(struct spi\_master \*master, struct spi\_device \*spi,struct spi\_transfer \*transfer);

/\*传输单个spi\_transfer实例数据\*/

void (\*handle\_err)(struct spi\_master \*master,struct spi\_message \*message);

int \***cs\_gpios**; /\*指向GPIO引脚编号数组，表示片选信号的GPIO编号\*/

/\* DMA channels for use with core dmaengine helpers \*/

struct dma\_chan \*dma\_tx; /\*DMA传输通道\*/

struct dma\_chan \*dma\_rx;

/\* dummy data for full duplex devices \*/

void \*dummy\_rx;

void \*dummy\_tx;

};

spi\_master结构体中主要成员简介如下：

●**list：**双链表成员，将spi\_master实例添加到全局双链表spi\_master\_list。

●**bus\_num：**主机控制器编号，从0开始编号，用于标识SPI主机控制器。

●**setup**：函数指针，用于修改传输速率和模式。

●**transfer：**函数指针，如果不为NULL，则不使用消息队列，直接调用此函数传输数据（已弃用）。

●**transfer\_one\_message：**函数指针，控制器调用此函数传输消息队列中单个消息（spi\_message实例）。

●**transfer\_one：**函数指针，在处理单个消息的函数中调用此函数传输单个spi\_transfer实例表示的数据。

●**queue**：双链表成员，表示消息队列，管理spi\_messages实例。

●**cs\_gpios**：指向GPIO引脚编号数组，表示片选信号的GPIO编号。

●**set\_cs：**函数指针，用于设置片选。

#### 2注册控制器

SPI主机控制器被视为设备，挂接到某一总线上，例如platform总线。主机控制器驱动需要定义控制器设备的驱动，如platform\_driver。板级相关文件内需要定义主机控制器设备，如platform\_devcie。在注册表示控制器的设备时将匹配驱动，匹配成功，调用驱动探测函数，函数内完成spi\_master实例的创建、初始化和注册。

内核提供的创建spi\_master实例的**spi\_alloc\_master()**函数定义在/drivers/spi/spi.c文件内：

struct spi\_master \*spi\_alloc\_master(struct device \*dev, unsigned size)

/\*dev：控制器父设备的device实例指针，不能为NULL，size：接在spi\_master之后的数据长度\*/

{

struct spi\_master \*master;

if (!dev) /\*父设备不能为NULL\*/

return NULL;

master = **kzalloc(size + sizeof(\*master), GFP\_KERNEL)**; /\*为实例分配内存空间\*/

if (!master)

return NULL;

device\_initialize(&master->dev); /\*初始化device成员\*/

master->bus\_num = -1;

master->num\_chipselect = 1;

master->dev.class = &**spi\_master\_class**; /\*设备类\*/

master->**dev.parent = get\_device(dev)**; /\*父设备\*/

**spi\_master\_set\_devdata(master, &master[1])**; /\*master[1]表示master实例之后的内存空间\*/

/\***master.dev.driver\_data=master[1]**，/include/linux/spi/spi.h\*/

return master;

}

spi\_alloc\_master()函数中dev参数表示spi\_master实例父设备的device实例指针，函数分配的内存空间前半部分为spi\_master实例，后半部分长度为size，表示主机控制器驱动私有数据。master.dev.driver\_data成员指向私有数据，表示驱动程序中使用的数据。

SPI主机控制器设备关联的设备类为spi\_master\_class（初始化函数中将注册），定义如下：

static struct class spi\_master\_class = {

.name = "spi\_master", /\*名称\*/

.owner = THIS\_MODULE,

.dev\_release = spi\_master\_release, /\*释放spi\_master实例的函数\*/

};

在创建spi\_master实例后，主机控制器驱动的probe()函数还需要对其进行进一步的初始化，然后将其注册到SPI控制器驱动框架中，注册函数为**spi\_register\_master()**，定义如下（/drivers/spi/spi.c）：

int spi\_register\_master(struct spi\_master \*master)

{

static atomic\_t dyn\_bus\_id = ATOMIC\_INIT((1<<15) - 1);

struct device \*dev = master->dev.parent; /\*父设备\*/

struct boardinfo \*bi;

int status = -ENODEV;

int dynamic = 0;

if (!dev)

return -ENODEV;

status = **of\_spi\_register\_master(master)**; /\*从设备树节点中提取片选引脚编号\*/

if (status)

return status;

if (master->num\_chipselect == 0)

return -EINVAL;

/\*如果实例中没有指定控制器编号，且存在设备树节点\*/

if ((master->bus\_num < 0) && master->dev.of\_node)

master->bus\_num = of\_alias\_get\_id(master->dev.of\_node, "spi"); /\*从设备树节点获取编号\*/

if (master->bus\_num < 0) { /\*如果master->bus\_num 小于0，则动态分配控制器编号\*/

master->bus\_num = atomic\_dec\_return(&dyn\_bus\_id);

dynamic = 1;

}

INIT\_LIST\_HEAD(&master->queue);

spin\_lock\_init(&master->queue\_lock);

spin\_lock\_init(&master->bus\_lock\_spinlock);

mutex\_init(&master->bus\_lock\_mutex);

master->bus\_lock\_flag = 0;

**init\_completion(&master->xfer\_completion)**; /\*初始化完成量\*/

if (!master->max\_dma\_len)

master->max\_dma\_len = INT\_MAX;

**dev\_set\_name**(&master->dev, "spi%u", master->bus\_num); /\*控制器设备名称：spiX\*/

**status = device\_add(&master->dev)**; /\*添加设备\*/

...

if (master->transfer) /\*master->transfer不为空表示不使用队列，直接由该函数完成数据传输\*/

dev\_info(dev, "master is unqueued, this is deprecated\n"); /\*已经弃用的方法\*/

else { /\*使用消息队列\*/

status = **spi\_master\_initialize\_queue(master)**;

/\*初始化主机控制器消息队列，见下一小节，/drivers/spi/spi.c\*/

...

}

mutex\_lock(&board\_lock);

**list\_add\_tail(&master->list, &spi\_master\_list)**; /\*spi\_master实例添加到全局双链表末尾\*/

**list\_for\_each\_entry(bi, &board\_list, list)**

**spi\_match\_master\_to\_boardinfo(master, &bi->board\_info)**; /\*/drivers/spi/spi.c\*/

/\*扫描全局boardinfo双链表，查找匹配项并创建spi\_device实例，见下文\*/

mutex\_unlock(&board\_lock);

**of\_register\_spi\_devices(master)**; /\*注册设备树中SPI总线设备\*/

acpi\_register\_spi\_devices(master);

done:

return status;

}

注册spi\_master实例函数的主要工作是初始化实例，创建消息队列（master->transfer为NULL），将实例添加到全局双链表末尾，查找匹配的spi\_board\_info实例，为匹配实例创建**spi\_device**实例并添加到SPI总线。另外，如果支持设备树，还需要扫描设备树spi控制器节点下设备节点，完成设备的添加。

总之，SPI主机控制器驱动探测函数主要工作是调用spi\_alloc\_master()函数创建spi\_master实例，然后对spi\_master实例进行设置（如函数指针成员等），最后调用spi\_register\_master()函数注册spi\_master实例。

### 8.7.3数据传输

SPI主机控制器需要提供数据传输的功能，数据传输接口函数调用主机控制器数据传输函数实现数据的传输，需要传输的数据由**spi\_message**结构体（SPI消息）封装，然后提交给主机控制器，由其负责数据的传输。

如果主机控制器定义了master->transfer()函数，则提交的消息由此函数负责传输，消息不由消息队列管理，这个方法已经弃用。如果主机控制器没有定义master->transfer()函数，则主机控制器会创建消息队列，由队列管理消息，控制器从队列中提取消息执行传输。

#### 1消息队列

SPI总线传输的数据由spi\_message结构体表示，通常提交的spi\_message实例插入到主机控制器管理的消息队列。在注册主机控制器时将初始化消息队列，控制器负责消息队列中消息的传输。

##### **■数据结构**

spi\_messages结构体定义如下（/include/linux/spi/spi.h）：

struct spi\_message {

struct list\_head **transfers**; /\*双链表头，管理spi\_transfers实例\*/

struct spi\_device **\*spi**; /\*指向请求消息传输的SPI设备（见下文）\*/

unsigned is\_dma\_mapped:1;

void (\*complete)(void \*context); /\*传输完成的回调函数\*/

void \*context; /\*complete回调函数的参数\*/

unsigned frame\_length; /\*消息总的传输数据长度\*/

unsigned actual\_length; /\*实际成功传输的数据长度，字节数\*/

int **status**; /\*成功为0，否则为负的错误码\*/

struct list\_head **queue**; /\*添加到主机控制器的消息队列\*/

void \*state; /\*驱动使用\*/

};

消息中实际传输的数据由spi\_transfer结构体表示，spi\_messages结构体中双链表头transfers管理着spi\_transfer实例。spi\_transfer结构体表示一段传输的数据，一个消息内可以由多段数据组成。

spi\_transfer结构体定义如下（/include/linux/spi/spi.h）：

struct spi\_transfer {

const void \***tx\_buf**; /\*发送缓存区指针\*/

void \***rx\_buf**; /\*接收缓存区指针\*/

unsigned  **len**; /\*缓存区长度，字节数\*/

dma\_addr\_t tx\_dma; /\*发送缓存区，DMA地址\*/

dma\_addr\_t rx\_dma; /\*接收缓存区，DMA地址\*/

struct sg\_table tx\_sg;

struct sg\_table rx\_sg;

unsigned cs\_change:1;

unsigned tx\_nbits:3;

unsigned rx\_nbits:3;

#define SPI\_NBITS\_SINGLE 0x01 /\* 1bit transfer \*/

#define SPI\_NBITS\_DUAL 0x02 /\* 2bits transfer \*/

#define SPI\_NBITS\_QUAD 0x04 /\* 4bits transfer \*/

u8 bits\_per\_word; /\*每个字的比特位数，为0则采用spi\_device的默认值\*/

u16 delay\_usecs;

u32 speed\_hz; /\*选择传输速率，为0则采用spi\_device的默认值\*/

struct list\_head **transfer\_list**; /\*双链表成员，添加到spi\_message实例中双链表\*/

};

以上数据结构组织关系如下图所示：



##### **■初始化消息队列**

在注册主机控制器的spi\_register\_master(struct spi\_master \*master)函数中将对主机控制器的消息队列进行初始化（前提条件是master->transfer为NULL），初始化函数为**spi\_master\_initialize\_queue(master)**，函数定义在/drivers/spi/spi.c文件内：

static int spi\_master\_initialize\_queue(struct spi\_master \*master)

{

int ret;

master->transfer = **spi\_queued\_transfer;** /\*初始化master->transfer函数指针成员，队列传输\*/

if (!master->transfer\_one\_message) /\*如果没有定义transfer\_one\_message()函数 \*/

master->transfer\_one\_message = spi\_transfer\_one\_message; /\*执行单个消息的传输\*/

ret = **spi\_init\_queue(master)**; /\*初始化消息队列，/drivers/spi/spi.c\*/

...

master->queued = true;

**ret =** **spi\_start\_queue(master)**; /\*激活消息队列，/drivers/spi/spi.c\*/

...

return 0;

...

}

spi\_master\_initialize\_queue()函数首先为控制器transfer和transfer\_one\_message函数指针成员赋值，请读者记住此处赋值的函数，在处理消息时需要调用这些函数；随后，调用**spi\_init\_queue(master)**函数初始化消息队列，最后调用**spi\_start\_queue()**函数激活对消息的处理，下面简要介绍一下这两个函数。

static int spi\_init\_queue(struct spi\_master \*master)

{

struct sched\_param param = { .sched\_priority = MAX\_RT\_PRIO - 1 };

master->running = false;

master->busy = false;

**init\_kthread\_worker(&master->kworker)**; /\*初始化内核工作者，/include/linux/kthread.h\*/

master->kworker\_task = kthread\_run(**kthread\_worker\_fn**, \

&master->kworker, "%s",dev\_name(&master->dev)); /\*创建内核线程\*/

...

**init\_kthread\_work(&master->pump\_messages, spi\_pump\_messages)**;

if (master->rt) { /\*如果控制器消息需实时处理\*/

dev\_info(&master->dev,"will run message pump with realtime priority\n");

sched\_setscheduler(master->kworker\_task, SCHED\_FIFO, &param);

/\*设置处理消息线程为实时线程\*/

}

return 0;

}

SPI消息的处理采用kthread\_worker机制（见第5章），简单地说就是建立内核工作者kthread\_worker实例，它通过双链表管理需要完成的工作kthread\_work实例，创建一个内核线程，线程内扫描kthread\_work实例链表，调用其中的func()函数，完成相关的工作。

这里使用的内核工作者是spi\_master.**kworker**，spi\_master.**pump\_messages**是工作kthread\_work实例，添加到工作者链表，其执行函数为**spi\_pump\_messages()，**spi\_master.**kworker\_task**指向内核线程结构实例。

spi\_start\_queue(master)函数用于将spi\_master.pump\_messages添加到工作者spi\_master.kworker中双链表，并激活创建的内核线程，由内核线程调用工作中的spi\_pump\_messages()函数处理消息队列中消息，函数源代码请读者自行阅读。

##### ■消息传输

在初始化消息队列时，spi\_master.pump\_messages工作中处理函数设为**spi\_pump\_messages()**，用于处理消息队列中消息，函数定义如下：

static void spi\_pump\_messages(struct kthread\_work \*work)

/\*work：指向工作实例\*/

{

struct spi\_master \*master =container\_of(work, struct spi\_master, pump\_messages);

**\_\_spi\_pump\_messages(master, true)**; /\*/drivers/spi/spi.c\*/

}

\_\_spi\_pump\_messages()函数代码简列如下：

static void \_\_spi\_pump\_messages(struct spi\_master \*master, bool in\_kthread)

{

unsigned long flags;

bool was\_busy = false;

int ret;

/\*锁定消息队列\*/

spin\_lock\_irqsave(&master->queue\_lock, flags);

/\*确保当前不在处理消息\*/

if (master->cur\_msg) {

spin\_unlock\_irqrestore(&master->queue\_lock, flags);

return;

}

/\* If another context is idling the device then defer \*/

if (master->idling) {

queue\_kthread\_work(&master->kworker, &master->pump\_messages);

spin\_unlock\_irqrestore(&master->queue\_lock, flags);

return;

}

/\*如果消息队列为空或当前主机控制器不在运行\*/

if (list\_empty(&master->queue) || !master->running) {

...

}

/\*从消息队列头部提取一个消息实例\*/

master->cur\_msg =**list\_first\_entry(&master->queue, struct spi\_message, queue)**;

**list\_del\_init(&master->cur\_msg->queue)**; /\*将消息从消息队列中移除\*/

if (master->busy)

was\_busy = true;

else

master->busy = true; /\*标记主机控制器忙\*/

spin\_unlock\_irqrestore(&master->queue\_lock, flags);

if (!was\_busy && master->auto\_runtime\_pm) {

ret = pm\_runtime\_get\_sync(master->dev.parent);

...

}

if (!was\_busy)

trace\_spi\_master\_busy(master);

if (!was\_busy && master->prepare\_transfer\_hardware) {

ret = master->**prepare\_transfer\_hardware**(master); /\*准备传输函数\*/

...

}

trace\_spi\_message\_start(master->cur\_msg);

if (master->prepare\_message) {

ret = master->prepare\_message(master, master->cur\_msg); /\*准备传输消息\*/

...

master->cur\_msg\_prepared = true;

}

ret = spi\_map\_msg(master, master->cur\_msg); /\*用于使用DMA的数据传输\*/

...

**ret = master->transfer\_one\_message(master, master->cur\_msg)**; /\*执行单个消息数据的传输\*/

...

}

\_\_spi\_pump\_messages()函数简单地说就是从控制器中取出第一个消息，调用**transfer\_one\_message()**函数完成单个消息的数据传输。

在前面初始化消息队列的函数中，如果主机控制器没有定义transfer\_one\_message()函数，将设为默认的**spi\_transfer\_one\_message()**函数，函数定义如下：

static int spi\_transfer\_one\_message(struct spi\_master \*master,struct spi\_message \*msg)

{

struct spi\_transfer \*xfer;

bool keep\_cs = false;

int ret = 0;

unsigned long ms = 1;

spi\_set\_cs(msg->spi, true); /\*设置片选信号\*/

/\*遍历spi\_message实例下的spi\_transfer实例双链表\*/

list\_for\_each\_entry(xfer, &msg->transfers, transfer\_list) {

trace\_spi\_transfer\_start(msg, xfer);

if (xfer->tx\_buf || xfer->rx\_buf) { /\*存在发送或接收缓存区\*/

reinit\_completion(&master->xfer\_completion); /\*初始化完成量\*/

ret = **master->transfer\_one(master, msg->spi, xfer)**; /\*传输单个spi\_transfer实例\*/

if (ret < 0) {

...

}

if (ret > 0) { /\*延时\*/

ret = 0;

ms = xfer->len \* 8 \* 1000 / xfer->speed\_hz;

ms += ms + 100; /\* some tolerance \*/

ms = wait\_for\_completion\_timeout(&master->xfer\_completion,msecs\_to\_jiffies(ms));

}

...

} else { /\*发送和接收缓存区都不存在\*/

...

}

...

} /\*遍历spi\_transfer实例链表结束\*/

...

**spi\_finalize\_current\_message(master)**; /\*/drivers/spi/spi.c\*/

/\*当前消息处理结束，将spi\_master.pump\_messages工作重新添加到工作者

\*spi\_master.kworker中双链，唤醒内核线程，调用mesg->complete()函数等。

\*/

return ret;

}

spi\_transfer\_one\_message()函数处理单个消息的流程是遍历spi\_message实例下的spi\_transfer实例双链表，对每个spi\_transfer实例调用master->transfer\_one(master, msg->spi, xfer)函数进行数据传输。

最后在spi\_finalize\_current\_message(master)函数中会将spi\_master.pump\_messages工作重新添加到工作者spi\_master.kworker中双链，唤醒内核线程，并调用mesg->complete()函数等。

#### 2接口函数

SPI控制器驱动为SPI设备驱动提供了数据传输的接口函数，设备驱动需要传输数据时，只需要调用接口函数，将需要传输的数据传递给控制器即可，由主机控制器实现数据的传输，设备驱动不必关心数据是如何实现传输的。

##### ■**写数据函数**

SPI主机控制器驱动提供的写数据函数为spi\_write()，定义如下（/include/linux/spi/spi.h）：

static inline int **spi\_write**(struct **spi\_device \*spi**, const void \*buf, size\_t len)

/\*spi：指向SPI设备，buf：缓存区指针，len：数据长度\*/

{

struct spi\_transfer t = { /\*创建spi\_transfer实例\*/

.tx\_buf = buf, /\*发送缓存区\*/

.len = len, /\*数据长度\*/

};

struct spi\_message m; /\*消息实例\*/

spi\_message\_init(&m); /\*初始化消息实例\*/

spi\_message\_add\_tail(&t, &m); /\*spi\_transfer添加到spi\_message链表末尾\*/

return **spi\_sync(spi, &m)**; /\*同步读写操作函数，见下文\*/

}

##### **■读数据函数**

SPI主机控制器驱动提供的读数据函数为spi\_read()，定义如下（/include/linux/spi/spi.h）：

static inline int **spi\_read**(struct **spi\_device \*spi**, void \*buf, size\_t len)

{

struct spi\_transfer t = {

.rx\_buf = buf,

.len = len,

};

struct spi\_message m;

spi\_message\_init(&m);

spi\_message\_add\_tail(&t, &m);

return **spi\_sync(spi, &m)**;

}

读写函数内部都是构建spi\_transfer和spi\_message实例，然后调用**spi\_sync(spi, &m)**函数完成数据的传输。

**spi\_sync()**函数定义如下（/drivers/spi/spi.c）：

int spi\_sync(struct spi\_device \*spi, struct spi\_message \*message)

{

return \_\_spi\_sync(spi, message, 0); /\*/drivers/spi/spi.c\*/

}

\_\_spi\_sync()函数定义在/drivers/spi/spi.c文件内，代码如下：

static int \_\_spi\_sync(struct spi\_device \*spi, struct spi\_message \*message, int bus\_locked)

{

DECLARE\_COMPLETION\_ONSTACK(done); /\*定义完成量\*/

int status;

struct spi\_master \*master = spi->master; /\*SPI主机控制器\*/

unsigned long flags;

status = \_\_spi\_validate(spi, message); /\*检测spi\_message实例有效性\*/

if (status != 0)

return status;

**message->complete = spi\_complete**; /\*传输完成的回调函数，唤醒在done完成量上等待的进程\*/

message->context = &done; /\*spi\_complete()函数参数\*/

message->spi = spi;

if (!bus\_locked)

mutex\_lock(&master->bus\_lock\_mutex);

if (**master->transfer == spi\_queued\_transfer**) {

/\*默认值，如果控制器驱动没有定义transfer()函数\*/

spin\_lock\_irqsave(&master->bus\_lock\_spinlock, flags);

trace\_spi\_message\_submit(message);

status = **\_\_spi\_queued\_transfer(spi, message, false)**; /\*将消息添加到消息队列尾部，返回0\*/

spin\_unlock\_irqrestore(&master->bus\_lock\_spinlock, flags);

} else { /\*主机控制器驱动定义了transfer()函数，弃用的方法\*/

status = spi\_async\_locked(spi, message); /\*异步执行数据传输，调用master->transfer()函数\*/

}

if (!bus\_locked)

mutex\_unlock(&master->bus\_lock\_mutex);

if (status == 0) { /\*处理消息\*/

if (master->transfer == spi\_queued\_transfer) /\*使用消息队列\*/

**\_\_spi\_pump\_messages(master, false)**; /\*处理消息队列，见上文\*/

**wait\_for\_completion(&done)**; /\*等待消息传输完成调用spi\_complete()函数唤醒当前进程\*/

status = message->status;

}

message->context = NULL;

return status;

}

\_\_spi\_sync()函数将spi\_message实例添加到主机控制器消息队列末尾，调用\_\_spi\_pump\_messages()函数执行消息的传输，这里首先处理的是消息队列的第一个消息，而不是本次提交的消息，处理完后会继续添加spi\_master.pump\_messages工作，处理队列中后面的消息。

调用spi\_sync()函数的进程将进入等待睡眠（同步传输），当提交的消息传输完成时，再唤醒当前进程。

内核在/include/linux/spi/spi.h头文件内还定义了创建、添加、同步消息的接口函数等，请读者自行阅读。

### 8.7.4控制器驱动示例

本小节以龙芯1B开发板为例，介绍其SPI主机控制器驱动的实现。龙芯1B芯片内置两个SPI主机控制器，外部引脚可复用。SPI主机控制器驱动套用了Bitbang框架，下面介绍SPI控制器驱动的实现。

#### 1 Bitbang框架

SPI控制器驱动中提供了Bitbang框架用于采用GPIO模拟SPI控制器的情形或某些SPI控制器，进行逐字或spi\_transfer实例的数据传输。Bitbang框架如下图所示：



使用Bitbang框架时，在调用spi\_alloc\_master()函数分配spi\_master实例时，在其后附上**spi\_bitbang**结构体实例。spi\_bitbang结构体中包含选择片选信号、实现spi\_transfer实例数据传输的函数指针等成员。spi\_master实例中的tranfer\_one\_message()函数赋值为**spi\_bitbang\_transfer\_one()**函数指针，而不是采用默认的函数，函数内调用spi\_bitbang结构体中的**txrx\_bufs()**函数完成消息数据的传输。

spi\_bitbang结构体定义在/include/linux/spi/spi\_bitbang.h头文件内：

struct spi\_bitbang {

spinlock\_t lock;

u8 busy;

u8 use\_dma;

u8 flags; /\*额外的spi->mode支持\*/

**struct spi\_master \*master**; /\*指向主机控制器spi\_master实例\*/

int (\*setup\_transfer)(struct spi\_device \*spi,struct spi\_transfer \*t); /\*更改时钟、字位数等\*/

void (\*chipselect)(struct spi\_device \*spi, int is\_on); /\*选择片选信号\*/

...

**int (\*txrx\_bufs)(struct spi\_device \*spi, struct spi\_transfer \*t)**; /\*发送接收消息函数\*/

/\* txrx\_word[SPI\_MODE\_\*]() just looks like a shift register \*/

u32 (\*txrx\_word[4])(struct spi\_device \*spi,unsigned nsecs,u32 word, u8 bits);

};

SPI控制器驱动（probe()函数）的主要工作是实现spi\_bitbang结构体中的函数，调用spi\_alloc\_master()函数创建结构体spi\_master和spi\_bitbang实例，并对其进行初始化，最后调用**spi\_bitbang\_start()**函数将创建的spi\_master实例注册到内核。

spi\_bitbang\_start()函数定义如下（/drivers/spi/spi-bitbang.c）：

int spi\_bitbang\_start(struct spi\_bitbang \*bitbang)

{

struct spi\_master \*master = bitbang->master; /\*SPI主机控制器\*/

int ret;

if (!master || !**bitbang->chipselect**) /\*必须定义chipselect()函数\*/

return -EINVAL;

spin\_lock\_init(&bitbang->lock);

if (!master->mode\_bits)

master->mode\_bits = SPI\_CPOL | SPI\_CPHA | bitbang->flags;

if (master->transfer || master->transfer\_one\_message) /\*不能手工设置这两个函数\*/

return -EINVAL;

master->prepare\_transfer\_hardware = spi\_bitbang\_prepare\_hardware; /\*默认的实现函数\*/

master->unprepare\_transfer\_hardware = spi\_bitbang\_unprepare\_hardware;

**master->transfer\_one\_message = spi\_bitbang\_transfer\_one**; /\*设置单个消息传输函数\*/

if (!bitbang->txrx\_bufs) { /\*若txrx\_bufs()函数指针成员为空\*/

bitbang->use\_dma = 0;

bitbang->txrx\_bufs = **spi\_bitbang\_bufs**; /\*赋值默认函数\*/

if (!master->setup) { /\*没有定义的函数，赋予默认值\*/

if (!bitbang->setup\_transfer)

bitbang->setup\_transfer =spi\_bitbang\_setup\_transfer;

master->setup = spi\_bitbang\_setup;

master->cleanup = spi\_bitbang\_cleanup;

}

}

ret = **spi\_register\_master(spi\_master\_get(master))**; /\*注册主机控制器spi\_master实例\*/

if (ret)

spi\_master\_put(master);

return 0;

}

spi\_bitbang\_start()函数中将对spi\_bitbang实例中各函数指针成员赋值，spi\_master实例中传输单个消息的函数设为**spi\_bitbang\_transfer\_one()**，函数定义如下：

static int **spi\_bitbang\_transfer\_one**(struct spi\_master \*master,struct spi\_message \*m)

{

struct spi\_bitbang \*bitbang;

unsigned nsecs;

struct spi\_transfer \*t = NULL;

unsigned cs\_change;

int status;

int do\_setup = -1;

struct spi\_device \*spi = m->spi; /\*SPI总线设备\*/

bitbang = **spi\_master\_get\_devdata(master)**; /\*master.dev.driver\_data=bitbang\*/

nsecs = 100;

cs\_change = 1;

status = 0;

list\_for\_each\_entry(t, &m->transfers, transfer\_list) { /\*扫描消息实例中的spi\_transfer实例链表\*/

/\* override speed or wordsize? \*/

if (t->speed\_hz || t->bits\_per\_word)

do\_setup = 1;

/\* init (-1) or override (1) transfer params \*/

if (do\_setup != 0) {

if (bitbang->setup\_transfer) {

status = **bitbang->setup\_transfer(spi, t)**; /\*启动数据传输\*/

if (status < 0)

break;

}

if (do\_setup == -1)

do\_setup = 0;

}

if (cs\_change) { /\*使能片选信号\*/

**bitbang->chipselect**(spi, BITBANG\_CS\_ACTIVE); /\*设置片选信号有效\*/

**ndelay(nsecs);** /\*延时100纳秒\*/

}

cs\_change = t->cs\_change;

if (!t->tx\_buf && !t->rx\_buf && t->len) {

status = -EINVAL;

break;

}

if (t->len) { /\*执行传输数据\*/

if (!m->is\_dma\_mapped)

t->rx\_dma = t->tx\_dma = 0;

**status = bitbang->txrx\_bufs(spi, t)**; /\*传输spi\_transfer实例中数据\*/

}

if (status > 0)

m->actual\_length += status;

if (status != t->len) {

if (status >= 0)

status = -EREMOTEIO;

break;

}

status = 0;

/\* protocol tweaks before next transfer \*/

if (t->delay\_usecs)

udelay(t->delay\_usecs);

if (cs\_change && !list\_is\_last(&t->transfer\_list, &m->transfers)) {

ndelay(nsecs);

bitbang->chipselect(spi, BITBANG\_CS\_INACTIVE); /\*消息传输完了，片选失效\*/

ndelay(nsecs);

}

} /\*消息内数据传输结束\*/

m->status = status;

if (!(status == 0 && cs\_change)) { /\*片选失效\*/

ndelay(nsecs);

bitbang->chipselect(spi, BITBANG\_CS\_INACTIVE);

ndelay(nsecs);

}

spi\_finalize\_current\_message(master); /\*完成消息数据传输完成的工作，见上文\*/

return status;

}

spi\_bitbang\_transfer\_one()函数遍历消息实例中的spi\_transfer实例链表，对每个spi\_transfer实例调用**bitbang->txrx\_bufs(spi, t)**函数执行数据的传输。

#### 2驱动程序示例

龙芯1B处理器具有两个SPI主机控制器，因此在板级文件中定义了两个对应的platform\_device实例：

static struct platform\_device ls1x\_spi0\_device = {

.name = "**spi\_ls1x**",

.id = 0,

.num\_resources = ARRAY\_SIZE(ls1x\_spi0\_resource),

.resource = ls1x\_spi0\_resource, /\*资源\*/

.dev = {

.platform\_data = &ls1x\_spi0\_platdata, //&ls1x\_spi\_devices，平台数据

},

};

static struct platform\_device ls1x\_spi1\_device = {

.name = "**spi\_ls1x**",

.id = 1,

.num\_resources = ARRAY\_SIZE(ls1x\_spi1\_resource),

.resource = ls1x\_spi1\_resource,

.dev = {

.platform\_data = &ls1x\_spi1\_platdata,//&ls1x\_spi\_devices,

},

};

在内核启动初始化子系统时会调用板级定义的初始化函数，初始化函数中会将platform\_device实例添加到platform总线。

SPI主机控制器驱动定义如下：

static struct platform\_driver ls1x\_spi\_driver = {

.probe = **ls1x\_spi\_probe**, /\*探测函数\*/

.remove = \_\_devexit\_p(ls1x\_spi\_remove),

.driver = {

.name = DRV\_NAME, /\*"spi\_ls1x"\*/

.owner = THIS\_MODULE,

.pm = NULL,

.of\_match\_table = ls1x\_spi\_match,

},

};

static int \_\_init ls1x\_spi\_init(void) /\*初始化函数\*/

{

return **platform\_driver\_register(&ls1x\_spi\_driver)**; /\*注册平台驱动\*/

}

**module\_init(ls1x\_spi\_init)**;

在添加/注册平台设备/驱动时会匹配总线上的驱动/设备，匹配成功将调用驱动的probe()函数，此处为**ls1x\_spi\_probe()**函数，函数内将会创建spi\_master和spi\_bitbang实例，并初始化，最后将spi\_master实例注册到内核。

龙芯1B处理器SPI主机控制器驱动程序中定义了如下数据结构：

struct ls1x\_spi {

**struct spi\_bitbang bitbang**; /\*必须是第一个成员\*/

struct completion done; /\*完成量\*/

void \_\_iomem \*base; /\*控制寄存器基地址\*/

int irq;

unsigned int div;

unsigned int speed\_hz;

unsigned int mode;

unsigned int len;

unsigned int txc, rxc;

const u8 \*txp;

u8 \*rxp;

#ifdef CONFIG\_SPI\_CS\_USED\_GPIO

unsigned int gpio\_cs\_count;

int \*gpio\_cs;

#endif

struct clk \*clk;

};

龙芯1B处理器SPI主机控制器驱动中数据结构组织关系如下图所示：



在控制器驱动的探测函数**ls1x\_spi\_probe()**中将创建以上数据结构实例，函数代码简列如下：

static int \_\_devinit ls1x\_spi\_probe(struct platform\_device \*pdev)

{

struct ls1x\_spi\_platform\_data \*platp = pdev->dev.platform\_data;

struct ls1x\_spi \*hw;

struct spi\_master \*master;

struct resource \*res;

int err = -ENODEV;

#ifdef CONFIG\_SPI\_CS\_USED\_GPIO

unsigned int i;

#endif

master = **spi\_alloc\_master(&pdev->dev, sizeof(struct ls1x\_spi))**;

/\*spi\_master实例之后是ls1x\_spi实例，且**master.dev.driver\_data=ls1x\_spi**\*/

if (!master)

return err;

/\* setup the master state. \*/

master->bus\_num = pdev->id;

master->num\_chipselect = 32;

master->mode\_bits = SPI\_CPOL | SPI\_CPHA | SPI\_CS\_HIGH;

master->setup = **ls1x\_spi\_setup**;

hw = **spi\_master\_get\_devdata(master)**; /\*指向spi\_master之后的ls1x\_spi实例\*/

**platform\_set\_drvdata(pdev, hw)**; /\*pdev->dev.driver\_data=ls1x\_spi\*/

/\* setup the state for the bitbang driver \*/

**hw->bitbang.master = spi\_master\_get(master)**; /\*指向主机控制器\*/

if (!hw->bitbang.master)

return err;

hw->bitbang.setup\_transfer = **ls1x\_spi\_setup\_transfer**; /\*启动数据传输\*/

hw->bitbang.chipselect = **ls1x\_spi\_chipselect**; /\*片选\*/

hw->bitbang.txrx\_bufs = **ls1x\_spi\_txrx\_bufs**; /\*实现spi\_transfer实例中数据传输的函数\*/

...

/\*设置并注册spi\_master实例等\*/

err = **spi\_bitbang\_start(&hw->bitbang)**;

...

return 0;

...

}

spi\_bitbang实例中的操作函数，通过操作SPI主机控制器的寄存器来实现，这里就不详细介绍了，有兴趣的读者可参考处理器手册阅读相关源代码。

### 8.7.5设备与驱动

前面介绍了SPI主机控制器的驱动，下面介绍SPI总线设备驱动，驱动框架如下图所示。

内核定义了SPI总线实例spi\_bus\_type，SPI总线设备由spi\_device结构体表示，SPI总线设备驱动由spi\_driver结构体表示。 

SPI设备的注册与前面介绍的平台总线设备的注册有所不同，板级相关代码中注册的是spi\_board\_info结构体实例。spi\_board\_info结构体中包含SPI设备使用的SPI主机控制器的信息，如使用SPI控制器的片选信号、最大传输速率等。

spi\_board\_info实例在内核中由双链表管理，在注册spi\_board\_info实例的**spi\_register\_board\_info()**函数中将查找spi\_board\_info实例匹配的spi\_master实例。若匹配成功将创建表示SPI总线设备的**spi\_device**实例（调用spi\_new\_device()函数），并关联到主机控制器spi\_master实例。spi\_device实例还将挂载到SPI总线，并查找匹配的设备驱动**spi\_driver**实例。若与驱动匹配成功将调用其中的probe()函数，在函数内将完成字符设备（或块设备）数据结构实例的创建和注册。

注册spi\_driver实例的函数为**spi\_register\_driver(sdrv)**，函数内调用SPI总线的匹配函数查找总线下匹配的spi\_device实例，匹配成功则调用spi\_driver实例的probe()函数，完成总线设备驱动程序的加载。

SPI总线设备驱动中文件操作结构中的函数调用SPI主机控制器驱动提供的接口函数，通过向控制器提交消息来完成数据的传输。在打开SPI总线设备文件时，将会建立file实例与表示设备的spi\_device实例之间的关联，以便在数据传输操作中将数据发送给正确的主机控制器和设备。

#### 1数据结构

SPI总线设备由spi\_device结构体表示，设备驱动由spi\_driver结构体表示，在初始化时注册了SPI总线实例。

##### **■设备与驱动**

SPI总线设备与驱动的数据结构定义如下（/include/linux/spi/spi.h）：

struct spi\_device {

struct device  **dev**; /\*内嵌device实例，将spi\_device实例添加到通用驱动模型\*/

struct spi\_master **\*master**; /\*指向主机控制器\*/

u32 max\_speed\_hz; /\*最大传输速率\*/

u8 **chip\_select**; /\*片选信号编号\*/

u8 bits\_per\_word;

u16 mode; /\*数据传输模式\*/

... /\*mode成员取值\*/

int **irq**; /\*中断编号\*/

void \*controller\_state; /\*主机控制器运行时状态\*/

void \*controller\_data; /\*板级相关定义的控制器信息\*/

**char modalias[SPI\_NAME\_SIZE]**; /\*匹配驱动的名称，与驱动spi\_device\_id列表匹配\*/

int cs\_gpio; /\*片选信号使用的GPIO编号（可选）\*/

};

SPI总线设备驱动数据结构定义如下：

struct spi\_driver {

const struct spi\_device\_id **\*id\_table**; /\*设备匹配列表，/include/linux/mod\_devicetable.h\*/

int (\***probe**)(struct spi\_device \*spi); /\*探测设备函数，由driver.probe()调用\*/

int (\*remove)(struct spi\_device \*spi); /\*解绑设备\*/

void (\*shutdown)(struct spi\_device \*spi); /\*关机回调函数\*/

struct device\_driver **driver**; /\*内嵌device\_driver实例\*/

};

spi\_device\_id结构体用于表示设备匹配列表，定义在/include/linux/mod\_devicetable.h头文件：

#define SPI\_NAME\_SIZE 32

#define SPI\_MODULE\_PREFIX "spi:"

struct spi\_device\_id {

**char name[SPI\_NAME\_SIZE]**; /\*匹配字符串\*/

kernel\_ulong\_t driver\_data; /\* Data private to the driver \*/

};

在匹配设备与驱动时，将比较spi\_device.modalias与spi\_driver.id\_table指向的列表成员，名称相同表示设备与驱动匹配成功。

##### **■总线**

SPI总线实例定义在/drivers/spi/spi.c文件内：

struct bus\_type spi\_bus\_type = {

.name = "spi",

.dev\_groups = spi\_dev\_groups, /\*默认的总线下设备属性\*/

.match = **spi\_match\_device**, /\*匹配函数，/drivers/spi/spi.c\*/

.uevent = **spi\_uevent**, /\*添加环境变量，/drivers/spi/spi.c\*/

};

SPI总线实例中没有定义探测probe()函数，在注册spi\_driver实例时会对其spi\_driver.driver.probe()函数赋值，所赋函数将调用spi\_driver.probe()函数。

下面主要看一下总线匹配函数，函数定义如下（/drivers/spi/spi.c）：

static int spi\_match\_device(struct device \*dev, struct device\_driver \*drv)

{

const struct spi\_device \*spi = to\_spi\_device(dev); /\*spi设备\*/

const struct spi\_driver \*sdrv = to\_spi\_driver(drv); /\*spi驱动\*/

/\* Attempt an OF style match \*/

if (of\_driver\_match\_device(dev, drv))

return 1;

/\* Then try ACPI \*/

if (acpi\_driver\_match\_device(dev, drv))

return 1;

if (sdrv->id\_table) /\*驱动设置了id\_table列表，则只匹配列表，不匹配驱动名称了\*/

return !!**spi\_match\_id(sdrv->id\_table, spi)**; /\*列表匹配\*/

return **strcmp(spi->modalias, drv->name)** == 0; /\*最后进行名称匹配，spi别名与驱动名称\*/

}

SPI总线匹配函数与平台总线匹配函数类似，主要是最后的列表匹配有所区别。

列表匹配函数spi\_match\_id()定义如下：

static const struct spi\_device\_id \*spi\_match\_id(const struct spi\_device\_id \*id,const struct spi\_device \*sdev)

{

while (id->name[0]) {

if (!**strcmp(sdev->modalias, id->name)**) /\*设备别名与驱动spi\_device\_id列表成员名称匹配\*/

return id;

id++;

}

return NULL;

}

spi\_match\_id()函数内将spi\_device结构体modalias[]成员与spi\_device\_id列表项成员的名称字符串比较，相同则表示匹配成功，不相同则不匹配。

如果驱动没有定义id\_table列表，最后会将spi\_device.modalias[]与驱动名称进行比较，相同表示匹配，不相同表示不匹配。

##### **■初始化**

内核在初始化后期调用spi\_init()函数对SPI总线驱动进行初始化，函数定义如下（/drivers/spi/spi.c）：

static int \_\_init spi\_init(void)

{

int status;

**buf = kmalloc(SPI\_BUFSIZ, GFP\_KERNEL)**; /\*申请缓存空间，/drivers/spi/spi.c\*/

...

status = **bus\_register(&spi\_bus\_type)**; /\*注册SPI总线\*/

if (status < 0)

goto err1;

status = **class\_register(&spi\_master\_class)**; /\*注册SPI主机控制器spi\_master\_class设备类\*/

if (status < 0)

goto err2;

if (IS\_ENABLED(CONFIG\_OF\_DYNAMIC))

WARN\_ON(of\_reconfig\_notifier\_register(&spi\_of\_notifier));

return 0;

...

}

**postcore\_initcall(spi\_init)**;

初始化函数内主要完成总线实例的注册和主机控制器设备类的注册。

#### 2注册设备信息

内核定义了spi\_board\_info结构体用于表示SPI设备与主机控制器的连接信息，一个控制器可连接多个SPI设备（通过片选信号确定）。

spi\_board\_info结构体定义如下（/include/linux/spi/spi.h）：

struct spi\_board\_info {

char **modalias[SPI\_NAME\_SIZE]**; /\*设备名称，赋予spi\_device实例，用于匹配驱动\*/

const void **\*platform\_data**; /\*平台数据结构，用于传递硬件信息，赋予spi\_device实例\*/

void \*controller\_data; /\*主机控制器信息，赋予spi\_device实例\*/

int irq; /\*中断编号\*/

/\* slower signaling on noisy or low voltage boards \*/

u32  **max\_speed\_hz**; /\*最大传输速率，赋予spi\_device实例\*/

u16  **bus\_num;**  /\*主机控制器（总线）编号，用于与spi\_master实例匹配\*/

u16 **chip\_select**; /\*片选信号，赋予spi\_device实例\*/

u16 mode; /\*赋予spi\_device实例\*/

};

内核在/drivers/spi/spi.c文件内定义了boardinfo结构体：

struct boardinfo {

struct list\_head list; /\*双链表成员，将boardinfo实例添加到全局双链表\*/

struct spi\_board\_info **board\_info**; /\*内嵌spi\_board\_info结构体成员\*/

};

boardinfo结构体内嵌spi\_board\_info结构体成员，list成员用于将实例添加到全局双链表board\_list。板级相关的代码中调用**spi\_register\_board\_info()**函数，向SPI驱动框架注册spi\_board\_info实例时，将会创建boardinfo实例，将spi\_board\_info实例复制到boardinfo实例board\_info成员；查找匹配的spi\_master实例，匹配成功将调用**spi\_new\_device()**函数为设备创建spi\_device实例，并添加到SPI总线（匹配驱动）。

spi\_register\_board\_info()函数定义如下（/drivers/spi/spi.c）：

int spi\_register\_board\_info(struct spi\_board\_info const \*info, unsigned n)

/\*info：指向spi\_board\_info实例数组，n：数组中实例数量（注册的实例数）\*/

{

struct boardinfo \*bi;

int i;

...

bi = **kzalloc(n \* sizeof(\*bi), GFP\_KERNEL)**; /\*创建boardinfo实例数组\*/

...

for (i = 0; i < n; i++, bi++, info++) {

struct spi\_master \*master;

**memcpy(&bi->board\_info, info, sizeof(\*info))**;

/\*复制spi\_board\_info实例至boardinfo实例数组\*/

mutex\_lock(&board\_lock);

**list\_add\_tail(&bi->list, &board\_list)**; /\*boardinfo实例添加到全局链表末尾\*/

**list\_for\_each\_entry(master, &spi\_master\_list, list)**  /\*扫描spi\_master链表找到匹配项\*/

**spi\_match\_master\_to\_boardinfo(master, &bi->board\_info)**; /\*匹配spi\_master实例\*/

mutex\_unlock(&board\_lock);

}

return 0;

}

注册spi\_board\_info实例函数内部创建boardinfo实例数组，从参数指向数组中复制数据至boardinfo实例数组，并将实例添加到全局双链表。对每个boardinfo实例扫描spi\_master双链表，对每个spi\_master实例调用spi\_match\_master\_to\_boardinfo()函数，判断是否匹配，若匹配成功则调用**spi\_new\_device()**函数创建并添加spi\_device实例。

**spi\_match\_master\_to\_boardinfo()**函数定义如下：

static void spi\_match\_master\_to\_boardinfo(struct spi\_master \*master,struct spi\_board\_info \*bi)

{

struct spi\_device \*dev;

if (**master->bus\_num != bi->bus\_num**) /\*两者总线编号不相同，则不匹配，返回\*/

return;

dev = **spi\_new\_device(master, bi)**; /\*总线编号相同，匹配，创建spi\_device实例\*/

if (!dev)

dev\_err(master->dev.parent, "can't create new device for %s\n",bi->modalias);

}

创建spi\_device实例的spi\_new\_device()函数定义如下：

struct spi\_device \*spi\_new\_device(struct spi\_master \*master,struct spi\_board\_info \*chip)

{

struct spi\_device \*proxy;

int status;

**proxy =** **spi\_alloc\_device(master)**; /\*分配spi\_device实例，并建立与master关联\*/

...

proxy->chip\_select = chip->chip\_select; /\*初始化spi\_device实例\*/

proxy->max\_speed\_hz = chip->max\_speed\_hz;

proxy->mode = chip->mode;

proxy->irq = chip->irq;

**strlcpy(proxy->modalias, chip->modalias, sizeof(proxy->modalias))**; /\*复制名称字符串\*/

**proxy->dev.platform\_data = (void \*) chip->platform\_data;** /\*板级数据结构的传递\*/

**proxy->controller\_data = chip->controller\_data**;

proxy->controller\_state = NULL; /\*从spi\_board\_info获取信息\*/

status = **spi\_add\_device(proxy)**; /\*向SPI总线添加设备，/drivers/spi/spi.c\*/

...

return proxy;

}

向SPI总线添加SPI设备的函数spi\_add\_device()定义如下：

int spi\_add\_device(struct spi\_device \*spi)

{

static DEFINE\_MUTEX(spi\_add\_lock);

struct spi\_master \*master = spi->master;

struct device \*dev = master->dev.parent;

int status;

/\*片选信号是否有效\*/

if (spi->chip\_select >= master->num\_chipselect) {

dev\_err(dev, "cs%d >= max %d\n",spi->chip\_select,master->num\_chipselect);

return -EINVAL;

}

**spi\_dev\_set\_name(spi)**; /\*设置device名称\*/

mutex\_lock(&spi\_add\_lock);

status = bus\_for\_each\_dev(&spi\_bus\_type, NULL, spi, spi\_dev\_check);

/\*遍历SPI总线中设备，调用spi\_dev\_check()函数，检查不能有重复的设备\*/

...

if (master->cs\_gpios) /\*GPIO引脚编号\*/

spi->cs\_gpio = master->cs\_gpios[spi->chip\_select];

status = **spi\_setup(spi)**; /\*设置片选信号，调用master实例setup()函数等，/drivers/spi/spi.c\*/

...

status = **device\_add(&spi->dev)**; /\*添加SPI设备，匹配spi\_driver成功则调用其probe()函数\*/

...

done:

mutex\_unlock(&spi\_add\_lock);

return status;

}

#### 3注册设备驱动

SPI总线设备驱动由spi\_driver实例表示，**spi\_register\_driver()**函数用于向内核注册驱动，函数定义如下：

int spi\_register\_driver(struct spi\_driver \*sdrv)

{

sdrv->driver.bus = &**spi\_bus\_type**; /\*设置总线实例\*/

if (sdrv->probe)

sdrv->driver.probe = **spi\_drv\_probe**; /\*设置默认函数，调用sdrv->probe()函数\*/

if (sdrv->remove)

sdrv->driver.remove = **spi\_drv\_remove**; /\*设置默认函数，调用sdrv->remove()函数\*/

if (sdrv->shutdown)

sdrv->driver.shutdown = **spi\_drv\_shutdown**; /\*设置默认函数，调用sdrv->shutdown()函数\*/

return **driver\_register(&sdrv->driver)**; /\*注册驱动实例\*/

}

在通用的注册驱动的driver\_register()函数中将在总线中查找匹配的设备实例，若匹配成功将调用驱动中spi\_driver.driver.probe()函数，进而调用spi\_driver.probe()函数完成字符/块设备驱动数据结构实例的创建和注册。

SPI总线设备驱动的一个例子是SD卡驱动程序（SPI接口），请读者参考第10章。

## 8.8 I2C总线

I2C总线是一种由Philips公司开发的两线式串行总线，用于连接处理器与外围设备。在物理结构上，I2C总线由数据线SDA和时钟SCL构成，每个器件都有一个唯一的识别地址。发送数据到总线的器件叫作发送器，从总线接收数据的器件称为接收器。初始化发送产生时钟信号和终止发送的器件叫作主机，被主机寻址的器件称为从机。

典型的I2C总线接口硬件架构如下图所示：



下面简单看一下I2C总线时序：

（1）开始/停止条件

如下图所示，在SCL处于高电平时，SDA线从高电平向低电平切换表示一个开始信号（开始数据传输）。在SCL处于高电平时，SDA线从低电平向高电平切换表示一个停止信号（停止数据传输）。



（2）数据传输

在数据传输过程中，数据线SDA上的数据在SCL时钟高电平周期内必须保持稳定，也就是说SCL为高电平时SDA线不能有电平切换，否则会被视为起始或停止条件。 SDA线只能在SCL为低电平时其电平才能切换。SCL脉冲可视为是对SDA线值的采样。

发送到SDA线上的每个字节必须是8bit。每次传输可以发送的字节数量不受限制，每个字节后必须跟一个响应位（ACK），由接收方发送到SDA线上。

下图示意了单字节写/读的时序：



读写数据前都需要发送器件地址和器件内部地址，前者表示寻址总线上哪个设备，后者寻址器件内部哪个字节。器件地址由8bit组成，高7bit表示器件地址，最低位表示读写位（高为读，低为写）。

写数据时先发送器件地址，再发送内部地址，最后发送数据，每个字节后需要有应答。读数据时先发送器件地址，再发送内部地址，然后还需要重新发送起始条件，再发送器件地址，最后才能从SDA线上读取数据，同样每个字节后也要有应答。

主机中的I2C控制器用于实现I2C总线的时序，实现给定数据的收发。I2C控制器可以是标准的控制器，也可以由GPIO模拟。

### 8.8.1概述

Linux内核中I2C主机控制器驱动、设备驱动与SPI总线都很相似，如下图所示：



I2C主机控制器作为一个设备挂接到平台platform总线上，在对应的平台设备驱动probe()函数中创建并初始化表示I2C主机控制器的**i2c\_adapter**结构体实例。i2c\_adapter结构体中包含I2C主机控制器在系统内的编号（来自于platform\_device）、deveice实例、控制器传输数据操作的**i2c\_algorithm**结构体实例等成员，i2c\_adapter实例将挂载到I2C总线i2c\_bus\_type实例上。

I2C总线设备由i2c\_client结构体表示，内嵌device实例，挂接到i2c总线上，并且主机控制器内嵌的deveice实例设为其父设备。i2c\_client实例并不是直接创建的，而是根据i2c\_board\_info实例创建的，板级相关代码需要创建i2c\_board\_info实例并调用接口函数i2c\_register\_board\_info()向内核注册（添加到全局双链表）。

在向内核注册主机控制器i2c\_adapter实例时，将扫描i2c\_board\_info实例（实际是i2c\_devinfo实例）双链表，对挂接到本控制器上的设备（由i2c\_board\_info实例表示）创建相应的**i2c\_client**实例，并挂接到i2c\_bus\_type总线上。

I2C设备驱动由i2c\_driver结构体表示，它与i2c\_client匹配。i2c\_client实例与i2c\_driver实例匹配上时将调用设备驱动中的probe()函数，函数内完成具体字符设备、块设备驱动数据结构实例的创建和注册。

设备驱动程序中定义的数据传输操作，通过I2C主机控制器驱动框架提供的接口函数完成，最终调用i2c\_client实例绑定i2c\_adapter实例关联的**i2c\_algorithm**实例中定义的函数完成数据传输。

I2C主机控制器驱动主要是创建i2c\_adapter和i2c\_algorithm实例，并向内核注册。在板级相关的代码中需要定义表示I2C总线设备的i2c\_board\_info实例并向内核注册，I2C总线设备驱动主要是实现i2c\_driver实例并向内核注册，其probe()函数负责设备驱动程序加载。

I2C驱动相关代码位于/drivers/i2c/目录下。

### 8.8.2 I2C总线

I2C总线实例中挂载了主机控制器、设备和设备驱动。本小节先介绍I2C总线实例的定义和初始化，后面小节介绍主机控制器、设备、设备驱动的定义和注册。

I2C总线实例定义在/drivers/i2c/i2c-core.c文件内：

struct bus\_type i2c\_bus\_type = {

.name = "i2c", /\*总线名称\*/

.match = **i2c\_device\_match**, /\*匹配函数，见下文\*/

.probe = **i2c\_device\_probe**, /\*注意总线定义了探测函数，见下文\*/

.remove = i2c\_device\_remove,

.shutdown = i2c\_device\_shutdown,

};

总线匹配函数和探测函数后面讲解总线设备和设备驱动时再作介绍。

下面先看一下I2C总线驱动的初始化，内核在启动阶段初始化子系统时调用i2c\_init()函数初始化I2C总线驱动，函数定义如下：

static int \_\_init i2c\_init(void)

{

int retval;

retval = of\_alias\_get\_highest\_id("i2c"); /\*I2C控制器ID编号最大值\*/

down\_write(&\_\_i2c\_board\_lock);

if (retval >= \_\_i2c\_first\_dynamic\_bus\_num)

\_\_i2c\_first\_dynamic\_bus\_num = retval + 1;

up\_write(&\_\_i2c\_board\_lock);

retval = **bus\_register(&i2c\_bus\_type)**; /\*注册总线实例，成功返回0\*/

if (retval)

return retval;

#ifdef CONFIG\_I2C\_COMPAT

i2c\_adapter\_compat\_class = class\_compat\_register("i2c-adapter");

if (!i2c\_adapter\_compat\_class) {

retval = -ENOMEM;

goto bus\_err;

}

#endif

retval = **i2c\_add\_driver(&dummy\_driver)**;

/\*向总线添加设备驱动dummy\_driver实例（此驱动什么也不做）\*/

if (retval)

goto class\_err;

if (IS\_ENABLED(CONFIG\_OF\_DYNAMIC))

WARN\_ON(of\_reconfig\_notifier\_register(&i2c\_of\_notifier));

return 0;

...

}

**postcore\_initcall(i2c\_init)**;

i2c\_init()函数主要是将i2c\_bus\_type实例注册到内核，并注册dummy\_driver设备驱动。

### 8.8.3 I2C主机控制器

系统中每个I2C主机控制器由i2c\_adapter结构体实例表示，挂接到i2c\_bus\_type总线上，i2c\_adapter结构体中包含实现数据传输的i2c\_algorithm结构体实例指针。

I2C主机控制器驱动主要工作是创建i2c\_adapter和i2c\_algorithm结构体实例，并将i2c\_adapter实例挂接到i2c\_bus\_type总线。

#### 1数据结构

I2C主机控制器i2c\_adapter结构体定义如下（/include/linux/i2c.h）：

struct i2c\_adapter {

struct module \*owner;

unsigned int **class**; /\*I2C总线类型编号，/include/linux/i2c.h\*/

const struct i2c\_algorithm **\*algo**; /\*指向控制器数据传输的数据结构\*/

void \*algo\_data; /\*i2c\_algorithm实例私有数据结构\*/

struct rt\_mutex bus\_lock; /\*实时互斥量\*/

int timeout; /\*超时时间（jiffies）\*/

int retries;

struct device **dev;** /\*内嵌device 实例\*/

**int nr**; /\*I2C控制器编号，系统内可能有多个I2C主机控制器\*/

char name[48]; /\*名称\*/

struct completion dev\_released; /\*完成量\*/

struct mutex userspace\_clients\_lock;

struct list\_head **userspace\_clients**; /\*链接表示总线下设备的i2c\_client实例\*/

struct i2c\_bus\_recovery\_info \*bus\_recovery\_info; /\*/include/linux/i2c.h\*/

const struct i2c\_adapter\_quirks \*quirks; /\*/include/linux/i2c.h\*/

};

i2c\_adapter结构体中主要成员简介如下：

●**class**：I2C总线类型，定义如下（/include/linux/i2c.h）：

#define I2C\_CLASS\_HWMON (1<<0) /\* lm\_sensors, ... \*/

#define I2C\_CLASS\_DDC (1<<3) /\* DDC bus on graphics adapters \*/

#define I2C\_CLASS\_SPD (1<<7) /\* Memory modules \*/

#define I2C\_CLASS\_DEPRECATED (1<<8)

●**algo：**i2c\_algorithm结构体指针，表示控制器数据传输的结构体，见下文。

●**algo\_data**：i2c\_algorithm结构体中操作函数使用的私有数据结构指针。

●**dev：**内嵌device实例，将i2c\_adapter实例挂接到I2C总线。

●**nr：**主机控制器编号，系统中可能存在多个I2C主机控制器，用于标识主机控制器（总线编号）。

●**userspace\_clients：**双链表成员，链接表示总线下设备的i2c\_client实例。

**i2c\_algorithm**结构体中主要包含主机控制器进行数据传输的函数指针成员，定义如下：

struct i2c\_algorithm {

int (**\*master\_xfer**)(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs,int num);

int (\*smbus\_xfer) (struct i2c\_adapter \*adap, u16 addr,unsigned short flags, char read\_write,

u8 command, int size, union i2c\_smbus\_data \*data);

u32 (\***functionality**) (struct i2c\_adapter \*); /\*返回控制器支持的特性\*/

#if IS\_ENABLED(CONFIG\_I2C\_SLAVE) /\*控制器可作为从设备\*/

int (\*reg\_slave)(struct i2c\_client \*client);

int (\*unreg\_slave)(struct i2c\_client \*client);

#endif

};

i2c\_algorithm结构体中主要操作函数指针成员简介如下：

●**functionality：**返回控制器支持的特性，返回的标记值定义在/include/uapi/linux/i2c.h头文件，例如：

#define I2C\_FUNC\_I2C 0x00000001

#define I2C\_FUNC\_10BIT\_ADDR 0x00000002

#define I2C\_FUNC\_PROTOCOL\_MANGLING 0x00000004 /\* I2C\_M\_IGNORE\_NAK etc. \*/

#define I2C\_FUNC\_SMBUS\_PEC 0x00000008

#define I2C\_FUNC\_NOSTART 0x00000010 /\* I2C\_M\_NOSTART \*/

#define I2C\_FUNC\_SLAVE 0x00000020

...

●**master\_xfer：**控制器作为主设备传输数据的函数，传输数据由i2c\_msg结构体表示，num表示i2c\_msg实例的数量，这是I2C主机控制器驱动需要实现的主要函数。

i2c\_msg结构体用于表示要传输的数据，结构体定义在/include/uapi/linux/i2c.h头文件：

struct i2c\_msg {

\_\_u16 **addr**; /\*从设备地址\*/

\_\_u16 flags; /\*标记，取值定义如下\*/

#define I2C\_M\_TEN 0x0010 /\*10位寻址，写数据\*/

#define I2C\_M\_RD 0x0001 /\*主设备自从设备读数据\*/

#define I2C\_M\_STOP 0x8000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_NOSTART 0x4000 /\* if I2C\_FUNC\_NOSTART \*/

#define I2C\_M\_REV\_DIR\_ADDR 0x2000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_IGNORE\_NAK 0x1000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_NO\_RD\_ACK 0x0800 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/

#define I2C\_M\_RECV\_LEN 0x0400 /\* length will be first received byte \*/

\_\_u16 len; /\*buf指向缓存区字节数量，须小于64KB\*/

\_\_u8 \***buf;** /\*读写数据缓存区指针\*/

};

#### 2数据传输

I2C主机控制器驱动提供了执行数据传输的接口函数，函数内调用i2c\_algorithm结构体中**master\_xfer()**函数传输数据，接口函数供具体I2C总线设备驱动程序调用。具体I2C总线设备驱动不需要关心数据是如何进行传输的，只需要调用接口函数传输数据即可。传输的数据由i2c\_msg结构体封装。

发送接收数据接口函数如下：

●int **i2c\_transfer**(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num)：传输由num个i2c\_msg实例组成的数据，返回传输的i2c\_msg实例数量，函数内调用adap->algo->master\_xfer(adap, msgs, num)函数。消息的传输没有进行排队处理，而是在持有锁（实时互斥锁）的情况下直接进行传输。

●int **i2c\_master\_send**(const struct i2c\_client \*client, const char \*buf, int count)：发送（写）数据，buf指向发送数据缓存区，count表示发送数据字节数（小于64KB），返回发送数据字节数。函数内构建一个消息i2c\_msg实例，调用i2c\_transfer()函数发送数据。

●int **i2c\_master\_recv**(const struct i2c\_client \*client, char \*buf, int count)：接收（读）数据，buf指向接收数据缓存区，count表示接收数据字节数（小于64KB），返回实际接收数据字节数。函数内构建一个消息i2c\_msg实例，调用i2c\_transfer()函数接收数据。

#### 3注册控制器

在板级相关的文件中需要为I2C主机控制器创建platform\_device实例，并向内核注册。用户需要在驱动程序目录/drivers/i2c/busses/下添加主机控制器驱动，驱动程序中主要是创建platform\_driver实例，并注册。platform\_driver实例的probe()函数中需要完成控制器硬件的初始化，创建和初始化i2c\_adapter实例，并向内核注册。另外还有一项最重要的工作就是实现i2c\_algorithm实例，定义其中的数据传输函数，并将实例指针赋予i2c\_adapter实例。

向内核注册i2c\_adapter实例的接口函数调用关系如下：



i2c\_add\_adapter()与i2c\_add\_numbered\_adapter()接口函数的区别在于前者添加的i2c\_adapter实例中未指定编号（nr成员值为-1），而由内核分配编号，后者添加的i2c\_adapter实例中指定了编号，并保存在nr成员中。两个接口函数最后都是调用**i2c\_register\_adapter**(struct i2c\_adapter \*adap)函数向内核注册主机控制器i2c\_adapter实例，函数代码简列如下（/drivers/i2c/i2c-core.c）：

static int i2c\_register\_adapter(struct i2c\_adapter \*adap)

{

int res = 0;

...

/\*安全性、有效性检查\*/

rt\_mutex\_init(&adap->bus\_lock);

mutex\_init(&adap->userspace\_clients\_lock);

INIT\_LIST\_HEAD(&adap->userspace\_clients); /\*初始化双链表成员\*/

if (adap->timeout == 0)

adap->timeout = HZ; /\*超时时间设为1s\*/

dev\_set\_name(&adap->dev, "i2c-%d", adap->nr); /\*设置内嵌device名称\*/

adap->dev.bus = &**i2c\_bus\_type;**  /\*总线类型\*/

adap->dev.type = &**i2c\_adapter\_type**; /\*设备类型，定义了一组默认属性\*/

res = **device\_register(&adap->dev)**; /\*注册设备\*/

if (res)

goto out\_list;

dev\_dbg(&adap->dev, "adapter [%s] registered\n", adap->name);

pm\_runtime\_no\_callbacks(&adap->dev);

#ifdef CONFIG\_I2C\_COMPAT

...

#endif

/\* bus recovery specific initialization \*/

if (adap->bus\_recovery\_info) {

struct i2c\_bus\_recovery\_info \*bri = adap->bus\_recovery\_info;

if (!bri->recover\_bus) {

dev\_err(&adap->dev, "No recover\_bus() found, not using recovery\n");

adap->bus\_recovery\_info = NULL;

goto exit\_recovery;

}

/\* Generic GPIO recovery \*/

if (bri->recover\_bus == i2c\_generic\_gpio\_recovery) {

if (!gpio\_is\_valid(bri->scl\_gpio)) {

dev\_err(&adap->dev, "Invalid SCL gpio, not using recovery\n");

adap->bus\_recovery\_info = NULL;

goto exit\_recovery;

}

if (gpio\_is\_valid(bri->sda\_gpio))

bri->get\_sda = get\_sda\_gpio\_value;

else

bri->get\_sda = NULL;

bri->get\_scl = get\_scl\_gpio\_value;

bri->set\_scl = set\_scl\_gpio\_value;

} else if (!bri->set\_scl || !bri->get\_scl) {

/\* Generic SCL recovery \*/

dev\_err(&adap->dev, "No {get|set}\_gpio() found, not using recovery\n");

adap->bus\_recovery\_info = NULL;

}

}

exit\_recovery:

/\* create pre-declared device nodes \*/

**of\_i2c\_register\_devices(adap)**; /\*查找设备树中传递的I2C设备创建并挂接i2c\_client实例\*/

acpi\_i2c\_register\_devices(adap);

acpi\_i2c\_install\_space\_handler(adap);

if (adap->nr < \_\_i2c\_first\_dynamic\_bus\_num)

**i2c\_scan\_static\_board\_info(adap)**;

/\*扫描i2c\_board\_info实例，为控制器下设备创建并挂接i2c\_client实例，见下一小节\*/

/\* Notify drivers \*/

mutex\_lock(&core\_lock);

**bus\_for\_each\_drv**(&i2c\_bus\_type, NULL, adap, **\_\_process\_new\_adapter**);

/\*遍历总线下驱动，调用\_\_process\_new\_adapter()函数探测驱动匹配的设备，见下一小节\*/

mutex\_unlock(&core\_lock);

return 0;

...

}

注册i2c\_adapter实例函数的主要工作是将表示控制器的adap->dev实例添加到i2c\_bus\_type总线，设备类型设为i2c\_adapter\_type；为连接到控制器的设备创建i2c\_client实例并添加到总线，设备信息可能来自设备树文件、i2c\_board\_info实例双链表等。I2C控制器下连接的设备由i2c\_client实例表示，实例是动态创建的，控制器设备为挂接设备的父设备。

#### 4控制器驱动示例

下面以龙芯1B开发板源代码中I2C主机控制器驱动为例说明I2C控制器驱动的实现（源代码来自于开发板资料）。I2C主机控制器驱动中定义了ls1x\_i2c结构体，如下：

struct ls1x\_i2c {

void \_\_iomem \*base; /\*控制寄存器基地址\*/

**struct i2c\_adapter adap**; /\*内嵌i2c\_adapter实例\*/

};

驱动程序中定义的i2c\_algorithm结构体实例如下，主要包含数据传输的函数：

static const struct i2c\_algorithm ls1x\_algorithm = {

.master\_xfer = **ls1x\_xfer**, /\*通过操作控制寄存器实现数据传输，请读者自行阅读源代码\*/

.functionality = ls1x\_functionality, /\*返回控制器特性\*/

};

I2C主机控制器驱动platform\_driver实例定义如下：

static struct platform\_driver **ls1x\_i2c\_driver** = {

.probe = **ls1x\_i2c\_probe**, /\*探测函数，初始化并注册I2C控制器实例\*/

.remove = \_\_devexit\_p(ls1x\_i2c\_remove),

.suspend = ls1x\_i2c\_suspend,

.resume = ls1x\_i2c\_resume,

.driver = {

.owner = THIS\_MODULE,

.name = "**ls1x-i2c**", /\*名称用于匹配platform\_device实例\*/

},

};

探测函数**ls1x\_i2c\_probe()**负责创建ls1x\_i2c结构体实例，并初始化和注册i2c\_adapter实例。

在内核初始化时将调用ls1x\_i2c\_init()函数注册主机控制器驱动**ls1x\_i2c\_driver**实例。

static int \_\_init ls1x\_i2c\_init(void)

{

return platform\_driver\_register(&ls1x\_i2c\_driver); /\*注册ls1x\_i2c\_driver实例\*/

}

arch\_initcall(ls1x\_i2c\_init); /\*内核初始化阶段调用此函数\*/

在板级相关的文件中需要定义表示I2C控制器的platform\_device实例并向内核注册，例如：

static struct platform\_device **ls1x\_i2c0\_device** = {

.name = "**ls1x-i2c**", /\*用于匹配驱动\*/

.id = 0, /\*编号0，还有另外两个I2C控制器\*/

.num\_resources = ARRAY\_SIZE(ls1x\_i2c0\_resource),

.resource = ls1x\_i2c0\_resource,

};

在添加platform总线设备和驱动时将查找匹配的驱动或设备，匹配成功将调用驱动的probe()函数为I2C主机控制器创建并注册i2c\_adapter实例。

下面简要看一下驱动探测函数ls1x\_i2c\_probe()的定义：

static int \_\_devinit ls1x\_i2c\_probe(struct platform\_device \*pdev)

{

struct ls1x\_i2c \*i2c;

struct resource \*res;

int rc;

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0); /\*内存资源，控制寄存器地址\*/

...

if (!(i2c = **kzalloc(sizeof(struct ls1x\_i2c), GFP\_KERNEL)**)) { /\*分配ls1x\_i2c实例\*/

...

}

if (!request\_mem\_region(res->start, resource\_size(res), "ls1x-i2c")) /\*申请资源\*/

return -EBUSY;

**i2c->base = ioremap(res->start, resource\_size(res));**  /\*控制寄存器基地址\*/

...

strlcpy(i2c->adap.name, "loongson1", sizeof(i2c->adap.name));

i2c->adap.owner = THIS\_MODULE;

i2c->adap.algo = &**ls1x\_algorithm**;

i2c->adap.class = I2C\_CLASS\_HWMON;

i2c->adap.algo\_data = i2c;

i2c->adap.dev.parent = &**pdev->dev**; /\*父设备\*/

i2c->adap.nr = pdev->id;

**ls1x\_i2c\_hwinit(i2c)**; /\*硬件初始化\*/

**platform\_set\_drvdata(pdev, i2c)**; /\*驱动数据指向ls1x\_i2c实例\*/

rc = **i2c\_add\_numbered\_adapter**(&i2c->adap); /\*注册i2c\_adapter实例\*/

...

return 0;

...

}

### 8.8.4设备与驱动

I2C总线设备由i2c\_client结构体表示，挂接到I2C总线上，其父设备是表示主机控制器的i2c\_adapter实例，设备驱动由i2c\_driver实例表示。i2c\_client与i2c\_driver匹配时将调用i2c\_driver内的probe()函数，注册设备驱动程序。

#### 1 I2C设备

在板级相关的文件中应当定义i2c\_board\_info结构体实例用于表示I2C总线设备并向内核注册（或通过设备树传递设备信息），在注册控制器i2c\_adapter实例时，将为连接到控制器的设备创建i2c\_client实例，并挂载到I2C总线。

##### ■添加设备信息

表示I2C总线设备的i2c\_client结构体定义如下（/include/linux/i2c.h）：

struct i2c\_client {

unsigned short **flags**; /\*标记\*/

unsigned short **addr;** /\*设备地址（低7bit）\*/

char **name**[I2C\_NAME\_SIZE]; /\*名称\*/

struct i2c\_adapter \***adapter**; /\*指向主机控制器i2c\_adapter实例\*/

struct device **dev**; /\*内嵌device实例\*/

int **irq;** /\*中断编号\*/

struct list\_head **detected**; /\*双链表成员，将实例添加到i2c\_driver实例中的双链表\*/

#if IS\_ENABLED(CONFIG\_I2C\_SLAVE)

i2c\_slave\_cb\_t slave\_cb; /\* callback for slave mode\*/

#endif

};

i2c\_client结构体主要成员简介如下：

●**flags：**标记，取值定义如下：

#define I2C\_CLIENT\_PEC 0x04 /\* Use Packet Error Checking \*/

#define I2C\_CLIENT\_TEN 0x10 /\* we have a ten bit chip address \*/

#define I2C\_CLIENT\_WAKE 0x80 /\* for board\_info; true iff can wake \*/

#define I2C\_CLIENT\_SCCB 0x9000 /\* Use Omnivision SCCB protocol \*/

i2c\_client结构体实例不是静态定义的，而是根据i2c\_board\_info结构体实例动态创建的。

板级相关的文件通过i2c\_board\_info结构体实例向内核传递I2C总线设备信息，i2c\_board\_info结构体定义如下（/include/linux/i2c.h）：

struct i2c\_board\_info {

char **type[I2C\_NAME\_SIZE]**; /\*名称，用于初始化**i2c\_client.name**成员，并匹配驱动\*/

unsigned short **flags**; /\*用于初始化i2c\_client.flags成员\*/

unsigned short **addr**; /\*赋予i2c\_client.addr成员，设备地址（有保留地址，不是从0开始）\*/

void \*platform\_data; /\*用于初始化i2c\_client.dev.platform\_data\*/

struct dev\_archdata \*archdata;

struct device\_node \*of\_node; /\*设备节点\*/

struct fwnode\_handle \*fwnode;

int **irq;** /\*中断编号，保存至i2c\_client.irq\*/

};

i2c\_board\_info表示设备的硬件信息，其中addr表示设备地址，I2C设备有一些保留的地址，不能使用，如下所示：

/\*

\* Reserved addresses per I2C specification:

\* 0x00 General call address / START byte

\* 0x01 CBUS address

\* 0x02 Reserved for different bus format

\* 0x03 Reserved for future purposes

\* 0x04-0x07 Hs-mode master code

\* **[0x08,0x77]** 可用的地址

\* 0x78-0x7b 10-bit slave addressing

\* 0x7c-0x7f Reserved for future purposes

\*/

在板级相关文件中需要定义i2c\_board\_info实例（数组），并调用接口函数**i2c\_register\_board\_info()**向内核注册，函数代码简列如下（/drivers/i2c/i2c-boardinfo.c）：

int \_\_init i2c\_register\_board\_info(int busnum,struct i2c\_board\_info const \*info, unsigned len)

/\*busnum：主机控制器编号，info：指向i2c\_board\_info数组，len：数组项数\*/

{

int status;

down\_write(&\_\_i2c\_board\_lock);

if (busnum >= \_\_i2c\_first\_dynamic\_bus\_num)

\_\_i2c\_first\_dynamic\_bus\_num = busnum + 1;

for (status = 0; len; len--, info++) {

**struct i2c\_devinfo \*devinfo**;

**devinfo = kzalloc(sizeof(\*devinfo), GFP\_KERNEL)**; /\*创建i2c\_devinfo实例\*/

...

devinfo->busnum = busnum;

**devinfo->board\_info = \*info**; /\*复制i2c\_board\_info实例至i2c\_devinfo实例\*/

**list\_add\_tail(&devinfo->list, &\_\_i2c\_board\_list)**; /\*i2c\_devinfo实例插入到全局双链表末尾\*/

}

up\_write(&\_\_i2c\_board\_lock);

return status;

}

注册i2c\_board\_info实例（数组）函数的主要工作是将i2c\_board\_info实例转换成i2c\_devinfo实例，并将i2c\_devinfo实例插入到全局双链表\_\_i2c\_board\_list末尾。

i2c\_devinfo结构体定义如下（/drivers/i2c/i2c-core.h）：

struct i2c\_devinfo {

struct list\_head list; /\*双链表成员\*/

int busnum; /\*控制器编号\*/

**struct i2c\_board\_info board\_info**; /\*i2c\_board\_info实例\*/

};

全局双链表\_\_i2c\_board\_list用于管理i2c\_devinfo实例。

##### ■创建i2c\_client

在注册主机控制器i2c\_adapter实例时将扫描\_\_i2c\_board\_list全局双链表，对挂接到本控制器的设备创建i2c\_client实例，并挂接到I2C总线。执行函数定义如下：

static void **i2c\_scan\_static\_board\_info**(struct i2c\_adapter \*adapter)

{

struct i2c\_devinfo \*devinfo;

down\_read(&\_\_i2c\_board\_lock);

list\_for\_each\_entry(devinfo, &\_\_i2c\_board\_list, list) {

if (**devinfo->busnum == adapter->nr**&& !**i2c\_new\_device(adapter,&devinfo->board\_info)**)

...

}

up\_read(&\_\_i2c\_board\_lock);

}

devinfo->busnum与adapter->nr相等表示设备连接在本控制器上，**i2c\_new\_device()**函数用于创建和注册表示设备的i2c\_client实例。

**i2c\_new\_device()**函数代码简列如下：

struct i2c\_client \*i2c\_new\_device(struct i2c\_adapter \*adap, struct i2c\_board\_info const \*info)

{

struct i2c\_client \*client;

int status;

client = **kzalloc(sizeof \*client, GFP\_KERNEL)**; /\*分配i2c\_client实例\*/

if (!client)

return NULL;

**client->adapter = adap**; /\*指向主机控制器i2c\_adapter实例\*/

**client->dev.platform\_data = info->platform\_data**;

if (info->archdata)

client->dev.archdata = \*info->archdata;

client->flags = **info->flags**;

client->addr = **info->addr**;

client->irq = **info->irq**;

strlcpy(client->name, **info->type**, sizeof(client->name)); /\*复制名称\*/

/\*设备地址有效性检查\*/

status = i2c\_check\_client\_addr\_validity(client);

...

/\* Check for address business \*/

status = i2c\_check\_addr\_busy(adap, client->addr);

if (status)

goto out\_err;

**client->dev.parent = &client->adapter->dev**; /\*父设备\*/

client->dev.bus = &i2c\_bus\_type; /\*总线\*/

client->dev.type = &**i2c\_client\_type**; /\*设备类型\*/

client->dev.of\_node = info->of\_node;

client->dev.fwnode = info->fwnode;

i2c\_dev\_set\_name(adap, client); /\*设置名称\*/

status = **device\_register(&client->dev)**; /\*注册设备\*/

...

return client;

...

}

注册I2C设备流程如下图所示（红色箭头）：



i2c\_register\_board\_info()函数只是将i2c\_devinfo实例添加到全局双链表，而没有创查找匹配的主机控制器，创建i2c\_client实例。在注册主机控制器i2c\_adapter实例时，将扫描i2c\_devinfo实例链表，创建并注册设备i2c\_client实例。是否意味着i2c\_devinfo实例的注册必须在i2c\_adapter实例的注册之前？

#### 2 I2C驱动

I2C设备驱动由i2c\_driver结构体表示，设备驱动程序中需要定义i2c\_driver实例并向内核注册。在向内核注册i2c\_driver实例时，将在I2C总线上查找匹配的i2c\_client实例，匹配成功调用i2c\_driver实例中的probe()函数加载设备的驱动程序。

I2C总线设备驱动由i2c\_driver结构体表示，结构体定义如下（/include/linux/i2c.h）：

struct i2c\_driver {

unsigned int class; /\*与adapter->class取值相同\*/

int (\*attach\_adapter)(struct i2c\_adapter \*) \_\_deprecated; /\*将被废弃的函数\*/

int (\***probe**)(struct i2c\_client \*, const struct i2c\_device\_id \*); /\*探测函数，必须定义\*/

int (\*remove)(struct i2c\_client \*); /\*解绑设备时调用的函数\*/

void (\*shutdown)(struct i2c\_client \*); /\*设备关机时调用的函数\*/

void (\*alert)(struct i2c\_client \*, unsigned int data); /\*改变传输协议等时调用的函数\*/

int (\*command)(struct i2c\_client \*client, unsigned int cmd, void \*arg);

struct device\_driver **driver**; /\*通用device\_driver实例\*/

const struct i2c\_device\_id **\*id\_table**; /\*匹配列表，用于驱动与设备匹配，必须定义\*/

int (\***detect**)(struct i2c\_client \*, struct i2c\_board\_info \*); /\*自动探测控制器下匹配的设备\*/

const unsigned short \***address\_list**; /\*自动探测设备地址数组\*/

struct list\_head **clients**; /\*链接自动探测的设备的i2c\_client实例\*/

};

i2c\_driver结构体中主要成员简介如下：

●**probe：**设备与驱动匹配时调用此函数，必须定义，完成设备驱动程序的加载。

●**detect：**用于自动探测主机控制器下的适用本驱动的设备，自动探测的设备不需要注册i2c\_board\_info 实例。

●**id\_table：**i2c\_device\_id结构体数组指针，用于匹配设备与驱动，必须定义，否则不能匹配设备。

i2c\_device\_id结构体定义在/include/linux/mod\_devicetable.h头文件，如下：

struct i2c\_device\_id {

char **name**[I2C\_NAME\_SIZE]; /\*名称字符，用于匹配设备\*/

kernel\_ulong\_t driver\_data; /\*驱动私有数据\*/

};

##### ■注册驱动

I2C设备驱动程序需要创建i2c\_driver实例，在probe()函数中完成设备驱动程序的注册，然后调用函数**i2c\_register\_driver()**向内核注册i2c\_driver实例，函数定义如下：

int i2c\_register\_driver(struct module \*owner, struct i2c\_driver \*driver)

{

int res;

...

driver->driver.owner = owner;

driver->driver.bus = **&i2c\_bus\_type**; /\*总线类型\*/

res = **driver\_register(&driver->driver)**; /\*注册驱动，成功返回0\*/

if (res)

return res;

pr\_debug("i2c-core: driver [%s] registered\n", driver->driver.name);

INIT\_LIST\_HEAD(&driver->clients);

**i2c\_for\_each\_dev(driver, \_\_process\_new\_driver)**;

/\*遍历总线设备调用**\_\_process\_new\_driver()**函数，自动探测控制器下设备\*/

return 0;

}

i2c\_register\_driver()函数主要是调用通用的driver\_register(&driver->driver)函数向内核注册驱动，然后扫描总线下设备，对主机控制器设备调用\_\_process\_new\_driver()函数，自动探测主机控制器下设备。

###### **●自动探测设备**

前面介绍过，在板级相关文件内需要定义并注册表示总线设备的 i2c\_board\_info实例，在注册主机控制器时，将根据i2c\_board\_info实例创建并注册表示设备的i2c\_client实例。

I2C设备驱动提供了自动探测控制器下设备的功能，不需要在代码中创建并注册i2c\_board\_info实例，而是在注册驱动时，对每个主机控制器调用**\_\_process\_new\_driver()**函数，在此函数中调用驱动中的**detect()**函数，探测主机控制器下本驱动适用的设备，为其动态创建i2c\_board\_info实例，进而创建并注册表示设备的i2c\_client实例。

\_\_process\_new\_driver()函数定义如下：

static int \_\_process\_new\_driver(struct device \*dev, void \*data)

/\*dev：I2C总线下设备，data：指向注册的i2c\_driver实例\*/

{

if (dev->type != **&i2c\_adapter\_type**) /\*不是表示主机控制器的设备，跳过\*/

return 0;

return **i2c\_do\_add\_adapter**(data, to\_i2c\_adapter(dev)); /\*对主机控制器调用此函数\*/

}

\_\_process\_new\_driver()函数只对主机控制器调用i2c\_do\_add\_adapter()函数，定义如下：

static int i2c\_do\_add\_adapter(struct i2c\_driver \*driver,struct i2c\_adapter \*adap)

/\*driver：指向i2c\_driver实例，adap：指向I2C控制器i2c\_adapter实例\*/

{

**i2c\_detect(adap, driver)**; /\*自动探测控制器下设备，创建并注册i2c\_client实例\*/

if (driver->attach\_adapter) { /\*将被废弃的函数\*/

...

}

return 0;

}

i2c\_detect()函数将调用i2c\_driver实例中的detect()函数，探测适用本驱动的设备（可以是多个设备），为其创建并注册i2c\_client实例，实例还添加到i2c\_driver->clients双链表，源代码请读者自行阅读。

#### 3总线匹配与探测函数

在注册表示总线的i2c\_client实例和表示驱动的i2c\_driver实例时，将会遍历驱动和设备，调用I2C总线匹配函数，检查两者的匹配性，若匹配，将会调用总线定义的probe()函数。

##### **■总线匹配函数**

i2c\_bus\_type总线匹配函数**i2c\_device\_match()**定义如下：

static int i2c\_device\_match(struct device \*dev, struct device\_driver \*drv)

{

struct i2c\_client \*client = i2c\_verify\_client(dev);

struct i2c\_driver \*driver;

if (!client)

return 0;

if (**of\_driver\_match\_device(dev, drv)**) /\*基于设备树的匹配函数\*/

return 1;

if (acpi\_driver\_match\_device(dev, drv))

return 1;

driver = **to\_i2c\_driver(drv)**; /\*指向i2c\_driver实例\*/

if (driver->id\_table)

return **i2c\_match\_id**(driver->id\_table, client) != NULL;

/\*i2c\_device\_id->name与client->name匹配\*/

return 0;

}

i2c\_match\_id()函数将id\_table指向的i2c\_device\_id数组项中的名称字符串与client->name名称比较，相同则表示匹配成功，函数返回i2c\_device\_id数组项指针，没有匹配的项返回NULL。

注意：在i2c\_match\_id()匹配函数中并没有比对i2c\_client和i2c\_driver名称，名称不做为检查匹配的条件，这与平台总线和SPI总线不同（这个总线最后会检查名称是否匹配）。

##### **■总线探测函数**

若设备i2c\_client与驱动i2c\_driver匹配成功，则将调用i2c\_bus\_type总线的探测函数，加载设备驱动程序。i2c\_bus\_type总线实例定义了探测函数，因此会优先调用它（否则调用device\_driver中探测函数），而平台总线和SPI总线实例没有定义探测函数。

I2C总线探测函数i2c\_device\_probe()定义如下：

static int i2c\_device\_probe(struct device \*dev)

{

struct i2c\_client \***client** = i2c\_verify\_client(dev);

struct i2c\_driver \***driver**;

int status;

if (!client)

return 0;

if (!client->irq) { /\*如果client->irq=0，获取中断号\*/

int irq = -ENOENT;

if (dev->of\_node)

irq = of\_irq\_get(dev->of\_node, 0);

else if (ACPI\_COMPANION(dev))

irq = acpi\_dev\_gpio\_irq\_get(ACPI\_COMPANION(dev), 0);

if (irq == -EPROBE\_DEFER)

return irq;

if (irq < 0)

irq = 0;

**client->irq = irq**;

}

**driver = to\_i2c\_driver(dev->driver)**; /\*指向i2c\_driver实例\*/

**if (!driver->probe || !driver->id\_table)** /\*必须定义的成员\*/

return -ENODEV;

if (!device\_can\_wakeup(&client->dev)) /\*/include/linux/pm\_wakeup.h\*/

device\_init\_wakeup(&client->dev,client->flags & I2C\_CLIENT\_WAKE);

/\*唤醒设备，/drivers/base/power/wakeup.c\*/

dev\_dbg(dev, "probe\n");

status = of\_clk\_set\_defaults(dev->of\_node, false);

if (status < 0)

return status;

status = dev\_pm\_domain\_attach(&client->dev, true);

if (status != -EPROBE\_DEFER) {

status = **driver->probe**(client, i2c\_match\_id(driver->id\_table,client));

/\*调用i2c\_driver中探测函数\*/

if (status)

dev\_pm\_domain\_detach(&client->dev, true);

}

return status;

}

i2c\_device\_probe()函数最终将调用i2c\_driver实例中的probe()函数，在此函数中将完成总线设备驱动程序的注册。

#### 4控制器驱动程序

主机控制器驱动程序是指将I2C主机控制器视为字符设备，导出为设备文件，用户进程通过对设备文件的读写，直接通过I2C总线实现数据的发送和接收。

用户进程通过/dev/i2c-x设备文件操作I2C主机控制器，x表示主机控制器编号，相关驱动程序代码位于/drivers/i2c/i2c-dev.c文件内（需选择I2C\_CHARDEV配置选项）。

I2C主机控制器驱动程序框架如下图所示，I2C主机控制器视为字符设备，主设备号为I2C\_MAJOR（256），从设备号为控制器在系统内的编号。

内核在/drivers/i2c/i2c-dev.c文件内定义了初始化函数i2c\_dev\_init()用于为I2C控制器创建并注册cdev实例（字符设备驱动数据结构），为每个控制器创建设备文件/dev/i2c-x，创建设备类i2c\_dev\_class实例等。

在打开主机控制器设备文件时，设备文件关联的file\_operations实例为i2cdev\_fops，其open()函数将为主机控制器创建i2c\_client实例，并关联到设备文件file实例。

i2cdev\_fops实例中读写函数调用主机控制器驱动提供的接口函数i2c\_master\_recv()/i2c\_master\_send()，完成总线数据的传输，相关源代码比较简单，请读者自行阅读。



## 8.9 USB总线

USB（Universal Serial BUS）通用串行总线，是由Compaq、DEC、IBM、Intel、NEC、Microsoft等公司于是1994年11月共同提出的，主要目的是为了制定统一的USB标准。USB设备使用方便，其文件传输速率快，而且具有热插拔性能，因此应用越来越广泛。常见的USB设备有USB鼠标、USB键盘、U盘、USB打印机等。

虽然作者水平有限，但还是想简要讲解一下USB总线规范，仅当抛砖引玉，然后介绍内核中USB总线和USB设备驱动框架的实现。USB规范文档可从www.usb.org网站下载。

USB总线及设备驱动源代码位于/drivers/usb/目录下，/drivers/usb/core/目录下包含USB总线驱动的核心代码，主要是主机控制器驱动的公共层代码，/drivers/usb/host/目录下是具体主机控制器的驱动，其它目录下是USB设备驱动程序。

### 8.9.1总线规范

USB总线规范已经到了USB4，由于龙芯1B处理器USB主机控制器只支持到USB2.0，更重要的是作者目前水平有限，因此本节仅限于讨论USB2.0。

#### 1 USB总线拓扑

USB总线使用4根电缆完成繁重要数据传输，如下图所示，它们分别是+5V电源线VBUS，地线GND、两根差分线D+和D-。主机端和设备端都有一个控制器，负责实现USB总线规范的数据传输。设备控制器负责总线与具体设备的数据传输，设备控制器通常是一个CPU（可带固件程序）。



USB总线通过集线器可构成分层星型拓扑结构，如下图所示：



图中集线器（HUB）是一类特殊的USB设备，是星型拓扑结构的中心，它类似于网络中的路由器，负责接收上层的数据，并分发给下一层设备。HUB通常有一个与上层连接的端口，多个连接下层设备的端口。主机控制器中嵌入一个HUB，称之为根HUB，主机通过根HUB提供多个端口。

通过集线器，USB总线最多能连接127个设备，含集线器，因为集线器也是设备，只不过是一种特殊的设备。总线上连接的每个设备有一个7位的总线地址，地址0默认给新接入的设备使用，因此设备实际可用的地址是127个。主机控制器通过设备地址确定与哪个设备通信。

USB2.0规范总共有3种数据传输速率：低速（Low Speed）1.5Mbit/s、全速（Full Speed）12Mbit/s、高速（High Speed）480Mbit/s（USB3.0还有超高速）。

HUB信号传输如下图所示，在数据向下游传输时，HUB采用广播的方式。在数据包被检测到后，为每个接通的下游端口建立一条数据通道，使其能够接收数据。接收下游数据时，则只接通发送数据的下游端口。



HUB必须支持3种速度模式。HUB在上游端口接入高速环境时，必须工作于高速模式，上游端口接入全/低速环境时，必须工作于全/低速模式。在上下游端口接入的是同速度模式设备时，HUB只是负责数据转发。当HUB工作于高速模式，下游接入全/低速设备时，HUB内部需通过事务转换器，将高速事务分割成全/低速事务，发送给下游设备。当HUB工作于全/低速模式时，下游设备只能工作于于全/低速模式，不能工作于高速模式。

USB设备逻辑视图如下图所示，一个USB设备可以有一个或多个配置状态。在任何时候只能有一个配置处于激活状态。

一个配置可以有一个或多个接口，在所处的配置被激活时，所有的接口也都处于激活状态。多重接口允许不同的主机方面的设备驱动与不同的USB设备的部分相关联。

接口可拥有一个或多个替代接口设置，任意时刻只能有一个接口设置被激活。一个接口设置下面拥有一个或多个端点。

设备、配置、接口、设置、端点都有对应的描述字，表示其信息，在主机枚举设备时（发送请求时）由设备发送给主机。



每个USB设备都提供了不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个）；每个配置则由多个接口组成；接口下有多个设置；设置由多个端点组成。

每个接口代表一个基本的功能，是USB设备驱动程序控制的对象，一个复杂的USB设备可以具有多个接口。通常说的USB设备驱动程序匹配的是USB接口。

端点是USB通信的最基本形式，对主机来说，每一个USB设备接口就是一组端点的集合。主机只有通过端点才能和设备进行通信，以使用设备的功能。在USB系统中每个端点都有独一无二的地址，该地址由设备地址和端点号组成。

端点就是一块典型的存储多个数据字节的缓存区。端点有一组属性，其中包括传输方向（IN/OUT）、传输类型（控制、中断、批量或等时）、总线访问频率、带宽、端点号和数据包的最大容量等。一个USB端点只能在一个方向承载数据，或从主机到设备（OUT端点），或从设备到主机（IN端点），因此端点传输是单向的。端点0是必须具有的端点，为控制端点，用于初始化设备参数。只要设备连接到USB上并且上电，端点0就可以被访问。端点1、2等一般用做数据端点，存放主机与设备间通信的数据。

#### 2传输类型

USB设备插入总线时，主机要枚举（检测）设备，获取设备各描述字，为设备分配地址等，这些后面再介绍，先简要介绍一下USB的数据传输类型，因为检测设备时就要用到数据传输。

USB数据传输类型共有4种，分别是控制传输、批量传输、中断传输和等时传输。每种类型都各具特点，适用于某种特殊的应用。

控制传输是USB规范所唯一规定的（必须的）传输类型，它能够使主机读取信息、设定设备地址并选取配置和采取其它安排。主机控制器枚举设备时使用的就是控制传输。

批量传输旨在那些对传输速率要求不严格的应用，如向打印机发送文件、从扫描仪接收数据或访问磁盘上的文件等工作。对于这些应用，快速传输当然更好，但若需要，数据还是可以等待的。若总线忙，则批量传输需要等待，而当总线空闲时，批量传输就是最快的传输类型。低速设备不支持批量传输端点。不要求设备一定支持批量传输，但某类设备需要这种传输。

中断传输用于那些必须周期性接收主机和其它设备动作的设备，或者用于短等待和短延时的设备。与控制传输不同，中断传输是低速设备用来传输数据的唯一方式。键盘和鼠标就使用中断传输来传递按键和鼠标动作数据。中断传输可使用任何速度模式。设备并不要求必须支持中断传输，但特定的设备可能会需要。

等时传输可保证传递时间，但不保证错误更正。使用等时传输的数据包括流音频和视频。等时传输是唯一在接到有效数据时不须回传的类型，因此使用这种传输必须允许偶尔出现错误。低速设备不支持等时传输。不要求设备必须支持等时传输，但特定设备会需要。

##### ■传输

前面介绍的4种传输类型中，每个USB传输由一个或多个事务组成，每个事务由一个令牌信息包，一个数据信息包（可能没有）和一个联络信息包组成，如下图所示。



每个信息包开头为PID（包标识符，8位）。令牌信息包中包含PID、7位设备地址、4位端点号和CRC（错误检查），数据信息包中包含PID、数据和CRC，联络信息包中只包含PID。

PID提供了有关事务的信息，其值如下表所示（低4位为PID，高4位为PID检测位）：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 信息包类型 | **PID**名字 | 取值 | 适用  传输类型 | 来源 | 总线  速率 | 描述 |
| 令牌信息包 | OUT | 0001 | 全部 | 主机 | 全部 | OUT事务，需要设备地址和端点号 |
| IN | 1001 | 全部 | 主机 | 全部 | IN事务，需要的设备地和端点号 |
| SOF | 0101 | 帧开始 | 主机 | 全部 | 帧开始标记和帧数（计数值） |
| SETUP | 1101 | 控制 | 主机 | 全部 | 用于设置事务，需要设备和端点地址 |
| 数据信息包 | DATA0 | 0011 | 全部 | 主机、设备 | 全部 | 偶数信息包PID |
| DATA1 | 1011 | 全部 | 主机、设备 | 全部 | 奇数信息包PID |
| DATA2 | 0111 | 等时 | 主机、设备 | 高速 | 数据信息包PID |
| MDATA | 1111 | 等时、分割事务 | 主机、设备 | 高速 | 数据信息包PID |
| 联络信息包 | ACK | 0010 | 控制、批量、  中断 | 主机、设备 | 全部 | 数据信息包接收无误（成功传输） |
| NAK | 1010 | 控制、批量、  中断 | 设备 | 全部 | 接收端不能接收数据，或者发送端 无法发送数据或无数据要发送 |
| STALL | 1110 | 控制、批量、  中断 | 设备 | 全部 | 控制请求不被支持或端点被停止 |
| NYET | 0110 | 控制写、批量  OUT、分割事  务 | 设备 | 高速 | 无误地接收了数据信息包，但还没准备好接收下一个；或者集线器还没有完成分割数据 |
| 特殊信息包 | PRE | 1100 | 控制、中断 | 主机 | 全速 | 主机发出的先行信号，表明下一个 信息包为低速（仅低速或全速线路） |
| ERR | 1100 | 全部 | 集线器 | 高速 | 由集线器返回，在分割事务（仅高速线路段上）中的低速或全速错误 |
| SPLIT | 1000 | 全部 | 主机 | 高速 | 先行于令牌信息包表明为分割事务 |
| PING | 0100 | 控制写、批量  OUT | 主机. | 髙速 | 批量/控制端点高速流探测 |
| EXT | 0000 | .无 | 主机 | 全部 | 协议拓展令牌 |

由上表可知，令牌信息包总是由主机发出的，也就是说每个事务总是由主机发起的，即使是中断传输也是由主机轮询发起的，而并是不像通常我们理解的设备硬件中断传输那样，由设备发给主机。

USB2.0主机控制器通过把时间在低速、全速模式下分成1ms宽的帧，在高速模式下分为125us的微帧方式来管理传输。主机将每个帧或微帧的一部分分配给各个传输，每个事务会在一个帧或微帧内完成而不被中断，并且不会有其它信息包插到单个事务中。

在设备端点描述字中指示了端点的传输类型，如控制（端点0）、批量、中断或等时，传输是通过端点来区分传输类型的，在传输的数据中并没有指示传输的类型。

##### ■控制传输

控制传输有两个用途：一是对于所有设备，主机通过控制传输向设备传递用于了解和配置设备的标准请求，设备返回描述字，二是控制传输还可传递任何由类或厂商定义的请求，设备返回类或厂商描述字。每个设备都必须通过端点0管道（默认）完成控制传输。控制传输是4种传输类型中最复杂的一种。

控制传输通常由3个阶段组成，分别是设置阶段、数据阶段和状态阶段，也可以没有数据阶段。数据阶段可以由一个或多个事务组成，设置和状态阶段包含一个事务，如下图所示。



###### ●**设置阶段**

设置阶段由一个事务组成，令牌信息包PID为**SETUP**。数据信息包PID为DATA0，数据长度固定为8个字节，分成5个字段，主要表示请求的类型和代码。请求如果正确接收，设备返回ACK。

数据信息包中8个字节的数据，5个字段分布如下图所示：



**●bmRequestType：**1个字节，位映射，确定数据流的方向、请求类型以及接收端。

bit7表示数据阶段数据流的方向，0表示（OUT事务）从主机到设备或者无数据阶段，1表示（IN事务）从设备到主机。

bit6到bit5表示请求类型：00表示USB标准请求之一、01表示专门的USB设备类定义的请求、10表示是厂商特定请求。

bit4到bit0表示请求接收者：00000表示请求由设备接收，00001表示由特定接口接收，00010表示由端点接收，00011表示设备中其他的元件。

●bRequest：1个字节，请求代码（名字），受bit6到bit5表示的请求类型影响。为00时，bRequest值为USB规范定义的11种标准请求；为01时，为USB设备类请求；为10时，为厂商特定请求。

●**wValue：**2个字节，不同请求可传递不同含义的信息。每个请求可按照它自己的方式定义这两个字节的意义。例如，在Set\_Address()请求中，wValue含有设备地址，在获取/设置描述字请求中，表示描述字类型和索引。

●**wlndex**：2个字节，因请求不同意义不同，一般用于传递索引或偏移量，例如指定接口或端点。传递端点索引时，比特位3到0规定了端点号；且比特位7等于0代表控制或者OUT端点，等于1代表IN端点。传递接口索引时，比特位7到0是接口号。所有未被定义的比特位都是0。

●**wLength**：2个字节，指定本控制传输第二个阶段，即数据阶段中传输数据的长度。对于从主机到设备的传输，wLength是主机打算传输的字节数。对于从设备到主机的传输，wLength是可取得的最大值， 设备应返回此数目或少些的字节。如果此字段是0，传输则没有数据阶段。

下表列出了对于USB规范定义的11种标准请求，以及字段的组合值：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **bmRequestType** | **bRequest**  **（请求）** | **wValue** | **wlndex** | **wLength** | 数据阶段  传输的数据 |
| 1000 0000B  1000 0001B  1000 0010B  （IN） | GET\_STATUS  （00h） | 0 | 0  接口号  端点号 | 2 | 设备、接口、端点状态 |
| 0000 0000B  0000 0001B  0000 0010B | CLEAR\_FEATURE  （01h） | 特征选择符 | 0  接口  端点 | 0 | 无 |
| 0000 0000B  0000 0001B  0000 0010B | SET\_FEATURE  （03h） | 特征选择符 | 0  接口号  端口号 | 0 | 无 |
| 0000 0000B | SET\_ADDRESS  （05h） | 设备地址 | 0 | 0 | 无 |
| 1000 0000B  （IN） | GET\_DESCRIPTOR  （06h） | 描述字类型和索引 | 0或语言ID | 描述字长度 | 描述字 |
| 0000 0000B | SET\_DESCRIPTOR  （07h） | 描述字类型和索引 | 0或语言ID | 描述字长度 | 描述字 |
| 1000 0000B  （IN） | GET\_CONFIGURATION  （08h） | 0 | 0 | 1 | 配置值 |
| 0000 0000B | SET\_CONFIGURATION  （09h） | 配置数值 | 0 | 0 | 无 |
| 1000 0001B  （IN） | GET\_INTERFACE  （0Ah） | 0 | 接口号 | 1 | 替换设置 |
| 0000 0001B | SET\_INTERFACE  （0Bh） | 替换设置 | 接口号 | 0 | 无 |
| 1000 0010B  （IN） | SYNCH\_FRAME  （0Ch） | 0 | 端点号 | 2 | 帧数目 |

下面对部分请求做简要介绍：

**（1）GET\_STATUS**：获取设备、接口、端点状态。

在数据阶段，设备返回2字节的状态值，对于设备、接口和端点，状态值意义不同。

设备：第一个字节的bit0表示设备是否自我供电，bit1表示设备是否支持远程唤醒，其它位保留，值为0。

接口：bit1和bit0应由设备定义，其它位为0。

端点：bit0表示当前端点是否停止（Halt），1表示停止，0表示可用。bit1可由设备定义，其它位为0。

**（2）CLEAR\_FEATURE/SET\_FEATURE：**清除/设置设备、接口或端点特征。

|  |  |  |  |
| --- | --- | --- | --- |
| 接收者  （**bmRequestType**） | 特征选择符（**wValue**） | 接收者索引（**wlndex**） | 描述 |
| 0000 0000B（设备） | DEVICE\_REMOTE\_WAKEUP（1） | 0 | 远程唤醒 |
| 0000 0001B（接口） | TEST\_MODE  （2） | 接口号 |  |
| 0000 0010B（端点） | ENDPOINT\_HALT  （0） | 端点号 | 暂停 |

（**3）SET\_ADDRESS：**设置设备地址。

wValue保存设备新地址，允许值为01h-7Fh，总线上每个设备，包括集线器，都有一个总线下唯一的地址。

（4）**GET\_DESCRIPTOR/SET\_DESCRIPTOR：**获取/设置设备描述字。

wValue保存描述字类型，wLength表示在描述字的长度。标准描述字类型如表所示：

|  |  |  |
| --- | --- | --- |
| **wValue** | **描述字类型** | **是否需要** |
| 0lh | 设备 | 是 |
| 02h | 配置 | 是 |
| 03h | 字符串 | 否，除非驱动程序要求，为可选的描述内容 |
| 04h | 接口 | 是 |
| 05h | 端点 | 是，使用端点0以外的端点 |
| 06h | 设备限定符 | 对于既支持全速又支持高速的设备，是。其他设备则否 |
| 07h | 其他速率配置 | 对于既支持全速又支持高速的设备，足.其他设备则否 |
| 08h | 接口功耗 | 否（推荐，但从未被普遍认可和实现） |
| 09h | OTG | 对于OTG设备，是 |
| 0Ah | 调试 | 否 |
| 0Bh | 接口联合 | 对某些复合设备，是 |
| 0Ch | 安全性 | 对于无线设备 |
| 0Dh | 密码钥匙 |
| 0Eh | 加密类型 |
| 0Fh | 二进制设备目标存储（BOS） | 对于超高速设备、无线设备以及支持链接功耗管理的设备，是 |
| 10h | 设备性能 |
| 11h | 无线端点伙伴 | 对于无线设备 |
| 30h | 超高速端点伙伴 | 对于超高速设备，是。对于其他设备，则否 |

在数据阶段的数据信息包中，将传递描述字的内容。每种类型的描述字由一个数据结构表示，所有数据结构中的第2个字节都是表示描述字类型的成员。具体类型描述字的定义这里就不介绍了，请读者自行参考相关资料。

下面对部分描述字的内容做简要介绍：

**设备描述字**：除复合设备外，所有设备有且仅有一个设备描述字，其中含有设备信息并确定了设备所支持的配置数量。设备描述字是设备连接到主机时主机读取的第一个描述字。设备描述字提供了关于设备、设备的配置以及任何设备所归属的类的信息。

**配置描述字**：每个配置，设有一个配置描述字，其中带有关于设备使用电源及配置所支持的接口数目的信息。设备接到对配置描述字的请求时，应当返回配置描述字，以及所有属于配置的接口、端点和其它多达所请求字节数的附属描述字。

**接口描述字**：每个接口，设备都有一个规定端点数目的接口描述字。接口描述字提供了关于设备所实现的功能和特性的信息。此描述字含有类、子类和协议的信息，以及接口所使用的端点数目。

接口可拥有一个或多个替代接口设置，任意时刻只能有一个接口设置被激活。每个设置含有一个接口描述字及其所需的附属描述字。接口描述字准确地说是接口设置描述字。

**端点描述字**：所有端点都有端点描述字，其中含有与端点通信所需的信息。没有端点描述字的接口，使用控制端点（0）完成全部通信，端点0从不会拥有描述字，因为所有设备必须包括端点0。

**字符串描述字**：可存储诸如制造商、设备名称或者序列号等文字信息。其它描述字可能含有指向字符串描述字的索引值。

**类和厂商描述字**：为设备或接口提供了结构化的方式，以给出关于功能的更详细的信息。

**（5）GET\_CONFIGURATION/SET\_CONFIGURATION：**获取/设置配置。

获取配置时，wValue值为0，在数据阶段返回设备配置值，每个配置值由一个整数表示。

设置配置时，wValue值表示设置设备配置。

**（6）GET\_INTERFACE/SET\_INTERFACE**：获取/设置接口。

wValue保存设置编号（设置接口时，否则为0），wlndex保存接口号，获取/设置接口的设置。

**（7）SYNCH\_FRAME：**帧同步请求。

wValue保存端点号，在数据阶段中的数据信息包中将传递帧号（2字节），在接到帧同步请求后，端点会返回一个帧号，新的次序从这个帧号开始。

此请求很少使用，因为很少需要它所提供的信息。

###### ●数据阶段

如果控制传输存在数据阶段，此阶段由一个或多个IN或OUT事务组成，如果在一个事务中不能传输所有要传输的数据则需要多个事务。在设置阶段数据信息包中的**wLength**字段，记录了数据阶段传输数据的长度。

数据阶段OUT事务如下所示（设置信息）：



令牌信息包PID为OUT，数据信息包PID为DATA1，若还有其它事务，则下一个事务中数据信息包PID为DATA0，再下一个又为DATA0，如此交替。联络信息包PID为ACK、NAK或STALL。

数据阶段IN事务如下所示（获取信息）：



设备将会向主机发送主机请求的信息，如果设备不能发送数据，或不支持此请求或端点停止，将直接发回NAK或STALL，而不会发送数据。

###### ●状态阶段

状态阶段由一个OUT或IN事务组成，用于报告整个控制传输是否成功。

控制写传输（设置信息）状态阶段事务如下图所示：



控制读传输（获取信息）状态阶段事务如下图所示：



状态阶段接收端发送零长数据信息包（数据长度为0）表示传输成功。

##### ■其它传输类型

批量传输由一个或多个IN或OUT事务构成，所有数据沿同一方向传输，如下图所示：



中断传输与批量传输的传输方式相同，不过中断传输并不由设备发起来，而是在主机轮询设备时发生，不同于设备触发的硬件中断。

等时传输更为简单，它不需要联络数据包，发送端不会检测接收方是否正确接收，传输方式如下图所示：



#### 3枚举设备

集线器的职责之一就是检测或移除处于下行端口处设备的连接。每个集线器都有一个中断IN端点，用来将这些事件报告给主机。系统启动时，集线器会通知主机是否有设备连接，包括额外的下行集线器和任何连接到那些（下行）集线器的设备。开机完成后，主机将继续**周期性地轮询**（USB 2.0），来获悉有关新连接或已移除设备的信息。

获悉新设备时，主机将向设备所属的集线器发送请求（新接入设备地址为0），使集线器建立一个处于主机和设备间的通信路径。然后主机会发布通向此设备、含有标准USB请求的控制传输，以尝试枚举此设备（获取设备信息）。所有的USB设备必须支持控制传输、标准请求和端点0。对于成功的枚举，设备必须通过返回被请求信息的形式响应请求，并执行其他所请求的动作。

USB 2.0规范定义了6个设备状态。枚举过程中，设备会先后经历开机（Powered）、缺省 （Default）、地址（Address）和配置（Configured）状态（另外两个状态是连接Attached状态和挂起Suspend状态）。在每个状态中，设备都定义了其性能和行为。

典型的USB2.0枚举顺序如下：

**（1）系统拥有了新设备**。用户将设备连接到USB端口，或者系统在连接有设备的情况下开机。端口可能处于主机内的根集线器上，或者处于连接主机和下行设备的其他集线器上。集线器提供端口电源，设备将处在开机状态（Powered）。设备可从总线获取100 mA的电流。

**（2）集线器检测设备**。集线器会监视其上每个端口的信号线（D+和D-）电压。集线器在每条线上都有一个14.25k~24.8kΩ的下拉电阻。设备还有900~1575Ω的上拉电阻，对于全速设备该电阻在D+上，对于低速设备则在D-上。高速设备是以全速来连接的。在连接到某个端口时，设备的上拉电阻会将它的信号线电位抬高，使集线器检测到设备连接。检测到设备时，集线器会继续提供电源，但还不会对设备发出USB传输。

**（3）主机获悉新设备**。每个集线器都会使用它的中断端点来报告集线器上的事件。报告只表明是集线器还是端口（如果是端口，是哪一个端口）经历了事件。当获悉事件时，主机将发送给集线器一个Get Port Status请求，以了解更多信息。Get Port Status请求和其他集线器请求都属于集线器所支持的标准请求。有新设备连接时，返回信息便会将此告知主机。

**（4）集线器检测设备是低速还是全速**。在复位（reset）设备之前，集线器通过检査两个信号线上的电压，确定设备是低速还是全速。集线器会确定闲置（idle）时哪条线有更高的电压，以此检测设备速度。然后，集线器将给主机发送信息，响应下一个Get Port Status请求。USB 1.x集线器可能会在总线复位之后检测设备速度。USB 2.0则要求在复位之前进行速度检测，以便使集线器在复位期间就知道(如下面所述的)被检测设备是否有高速能力。

**（5）集线器复位设备**。当主机获悉一新设备时，主机将向集线器发送一个Set Port Feature请求，此信号要求集线器复位端口。集线器会将设备的USB数据线置于复位情况下（至少10ms）。复位是一种特殊的情况，这时D+和D-都为逻辑低（正常情况下，这两条线有相反的逻辑状态）。集线器只对新设备发送复位。总线上的其他集线器和设备都不会看到此复位信号。

**（6）主机了解全速设备是否支持高速状态**。检测设备是否支持高速状态使用两个特殊信号状态。在Chirp J状态只有D+线被驱动，而在Chirp K状态只有D-线被驱动。

复位期间，支持高速状态的设备会发送Chirp K信号。高速集线器检测到该Chirp K信号后，会响应一串交替的Chirp K与Chirp J。当检测到KJKJKJ的样式时，设备会移除它的全速上拉电阻，然后以高速状态执行接下来的通信。如果集线器没有对设备的Chirp K信号作出响应，设备便知道它必须以全速状态继续后面的通信。所有的高速设备都必须能够响应全速状态下的控制请求。

**（7）集线器在设备和总线间建立一条信号路径**。主机会通过发送Get Port Status请求证明设备已经摆脱了复位状态。返回数据中的一个数据位将用于表明设备是否仍处于复位状态。若有必要，主机还会重复此请求直到设备离开复位状态。

当集线器的复位信号被移除时，设备便处于缺省状态（Default）。设备的USB寄存器都处于复位状态，且设备已准备好响应端点0处的控制传输。设备会使用缺省地址**00h**来与主机通信。

**（8）主机发送Get Descriptor请求以了解缺省管道的最大信息包尺寸**。主机向设备地址00h，端点0发送此请求。由于主机每次只枚举一个设备，故也只有一个设备会响应发往设备地址00h的通信，即使此时有多个设备连接。设备描述字的第8字节含有端点0所支持的最大信息包尺寸。

**（9****）主机指定一个地址**。复位完成时，主机控制器将通过发送Set Address请求，为设备指定唯一的地址。设备将以缺省地址完成请求的状态阶段，之后才开始执行新地址。设备现在处于地址状态（Address）。从这个端口所发出的所有通信都会指向新地址。地址会一直有效，直到设备断开连接、集线器复位端口或系统重启。下一次枚举时，主机可能会分给设备一个不同的地址。

**（10）主机了解设备能力**。主机向新地址发送Get Descriptor请求，读取设备描述字。这一次主机会取回整个描述字。描述字含有端点0的最大信息包尺寸、设备支持的配置数量，以及其他的设备基本信息。

主机会通过请求一个或多个指定在设备描述字内的配置描述字，继续了解设备。对配置描述字的请求，实际上就是对配置描述字和其后跟着它多达请求字节数的全部附属描述字的请求。

设备会通过发送配置描述字来做出响应，其后会跟有全部配置的附属描述字（包括接口描述字）。 而接口描述字后都跟着接口的所有端点描述字。有些配置也会有类或厂商专属描述字。

**（11）主机指定并加载设备驱动程序（复合设备除外）**。在根据描述字了解完设备信息后，主机会寻找最匹配的驱动程序，来管理与设备的通信。Windows主机使用INF文件确认最优匹配。INF文件可能是USB类的系统文件，或者厂商提供的含有厂商ID和产品ID的文件。

复合设备对这一顺序来说是个例外，它可在配置中含有指定到多个接口的不同驱动程序。主机只能在接口被使能后才可指定这些驱动程序，因此主机必须首先按下面所述的过程进行设备配置。

**（12）主机的设备驱动程序会选择配置。**从描述字那里了解完设备信息后，设备驱动程序会通过发送带有所需配置号的Set Configuration请求，来请求某一配置。许多设备只支持一种配置。若设备支持多个配置，驱动程序便可根据驱动程序所含有的设备信息，决定请求哪一配置；或者驱动程序也可询问用户或直接选择其中的第一个配置（许多驱动程序只会选择第一个配置）。接到请求时，设备会实现那个被请求的配置。至此设备就进入了（己）配置状态，设备的接口也就被使能了。

对于复合设备，主机现在就可以指定驱动程序了。而其他设备，主机使用从设备取得的信息来为每一个在配置中被激活了的接口找到驱动程序。之后，设备即为可用。

集线器也属于USB设备，且主机会以与其他设备相同的方式枚举新连接的集线器。如果集线器上还连接有设备，主机会在集线器通知主机它的存在后，对这些设备进行枚举。

**连接状态**：如果集线器不给设备的VBUS线提供电源，设备便处于连接状态（Attached）。若集线器检测到“过载电流”情况，或者主机请求集线器从此端口移除电源，不供给电源的情况就会发生。VBUS上没有电源，主机和设备就不能通信，因此从它们的角度看，这与设备没有连接的情况是一样的。

**挂起状态**：设备会在检测到至少3ms没有总线动作的情况下进入挂起状态（Suspend），这里的总线动作也包括帧开端标记（SOF）。在挂起状态，设备应限制它对总线功耗的使用。配置与没配置过的设备都必须支持这种状态。

### 8.9.2驱动框架

USB总线及设备驱动程序框架如下图所示。

USB主机控制器主要是负责实现主机与设备（端点）之间的数据传输。主机控制器由usb\_hcd结构体表示，其关联的hc\_driver结构体中的函数主要就是实现主机与设备（端点）之间的数据传输（按照USB规范执行）。USB驱动中传输数据由urb结构体封装，后面将介绍urb的处理流程。

主机控制器驱动的主要工作就是实现hc\_driver实例，调用内核提供的接口函数创建和添加usb\_hcd实例。

内核定义了USB总线实例usb\_bus\_type。USB设备由usb\_device结构体表示，结构体中包含设备配置、接口、端点（描述字）等信息。

USB主机控制器中内嵌一个根HUB，HUB也被视为USB设备。因此在添加usb\_hcd实例时，将创建并注册表示根HUB的usb\_device实例，挂载到USB总线上。

USB设备usb\_device实例将与USB设备驱动usb\_device\_driver实例匹配。内核定义并注册了通用设备驱动usb\_generic\_driver实例。在其probe()函数中将选择设备配置，获取设备接口信息，为接口创建并注册usb\_interface实例。usb\_interface实例也挂载到USB总线，与之匹配的是接口驱动usb\_driver实例，在其probe()函数中将加载接口驱动程序（请读者注意区分设备与接口，设备驱动与接口驱动）。

HUB接口对应的驱动是hub\_driver实例，其probe()函数将为HUB创建、设置usb\_hub结构体实例，并激活HUB。usb\_hub结构体中包含HUB端口信息等。



USB总线传输的数据由urb结构体封装，内核提供了分配、填充和提交urb实例的接口函数。提交的urb被添加到端点的urb实例双链表中，由主机控制器关联hc\_driver实例中的函数负责数据传输（需要进行调度）。传输完成后，urb实例被移动到usb\_hcd实例中的urb实例双链表，通过tasklet机制处理传输完成的urb实例，主要是调用实例中的complete()函数。



在创建HUB对应的usb\_hub实例时，将创建一个用于中断传输的urb实例，用于探测设备。主机控制器usb\_hcd实例中内嵌一个定时器（周期触发），定时器到期函数将检测根HUB是否有状态变化，如果有则将根HUB关联的中断传输urb添加到usb\_hcd中的传输完成urb实例双链表。其complete()函数将检测HUB端口的变化，枚举设备，为设备创建并添加usb\_device实例。usb\_device实例将匹配通用USB设备驱动，它为设备选择配置，获取接口信息，创建并注册usb\_interface实例，并查找匹配的接口驱动usb\_driver实例，在其probe()函数中将加载用户实现的设备（接口）驱动程序，这与前面介绍的根HUB设备的创建相似。

### 8.9.3主机控制器

USB主机控制器主要负责主机与设备的数据传输。

USB主机控制器作为设备挂接到某个总线上，例如：platform总线。主机控制器由usb\_hcd结构体表示，在主机控制器设备匹配驱动的probe()函数中需要为控制器创建并添加usb\_hcd实例，内核提供了创建和添加usb\_hcd实例的接口函数。主机控制器驱动程序还有一项重要的工作就是实现hc\_driver实例，它将赋予usb\_hcd实例。

#### 1数据结构

USB总线bus\_type结构体实例usb\_bus\_type，在内核中可认为是表示逻辑上的USB总线，即系统中所有的USB设备及驱动都挂接到此总线。系统内可能有多个USB主机控制器，即有多个USB物理总线。

每个主机控制器由usb\_hcd结构体表示，其中内嵌的usb\_bus结构体成员可认为是表示物理上的USB总线。usb\_hcd结构体关联的hc\_driver结构体主要是操作函数指针，用于执行数据传输等操作。

下面将介绍usb\_hcd、usb\_bus和hc\_driver结构体的定义。

##### ■usb\_hcd

USB主机控制器在内核中由**usb\_hcd**结构体表示，定义在/include/linux/usb/hcd.h头文件：

struct usb\_hcd {

struct usb\_bus **self**; /\*表示物理总线\*/

struct kref kref; /\*引用计数\*/

const char \*product\_desc; /\*product/vendor string（厂商描述字符串）\*/

int speed; /\*根HUB速率\*/

char irq\_descr[24]; /\*driver + bus #\*/

struct timer\_list **rh\_timer**; /\*根HUB轮询定时器\*/

struct urb \*status\_urb; /\*用于传递根HUB状态变化的urb\*/

#ifdef CONFIG\_PM

struct work\_struct wakeup\_work; /\*为远程唤醒\*/

#endif

/\*硬件信息和状态\*/

const struct hc\_driver **\*driver**; /\*由具体主机控制器驱动实现的操作接口，hc\_driver实例\*/

struct usb\_phy \***usb\_phy**; /\*OTG，/include/linux/usb/phy.h\*/

struct phy \*phy;

unsigned long **flags**; /\*标记成员，取值，见下文\*/

/\*HCD注册和移除时设置的标记\*/

unsigned rh\_registered:1; /\*根HUB是否已经注册了\*/

unsigned rh\_pollable:1; /\*是否允许轮询根HUB\*/

unsigned msix\_enabled:1; /\* driver has MSI-X enabled? \*/

unsigned remove\_phy:1; /\* auto-remove USB phy \*/

unsigned uses\_new\_polling:1;

unsigned wireless:1; /\*无线HCD\*/

unsigned authorized\_default:1;

unsigned has\_tt:1; /\* Integrated TT in root hub \*/

unsigned amd\_resume\_bug:1; /\* AMD remote wakeup quirk \*/

unsigned can\_do\_streams:1; /\* HC supports streams \*/

unsigned tpl\_support:1; /\* OTG & EH TPL support \*/

unsigned cant\_recv\_wakeups:1;

unsigned int **irq**; /\*中断编号\*/

void \_\_iomem \***regs**; /\*控制寄存器地址\*/

resource\_size\_t **rsrc\_start**; /\*memory/io resource start\*/

resource\_size\_t **rsrc\_len**; /\*memory/io resource length\*/

unsigned power\_budget; /\* in mA, 0 =无限制 \*/

struct giveback\_urb\_bh **high\_prio\_bh;**  /\*giveback\_urb\_bh结构体，/include/linux/usb/hcd.h\*/

struct giveback\_urb\_bh **low\_prio\_bh**;

struct mutex \*bandwidth\_mutex;

struct usb\_hcd \*shared\_hcd;

struct usb\_hcd \*primary\_hcd;

#define HCD\_BUFFER\_POOLS 4

struct dma\_pool \***pool[HCD\_BUFFER\_POOLS]**; /\*缓存（32，128，512，2048），/mm/dmapool.c\*/

int **state**; /\*状态，取值见下文\*/

unsigned long **hcd\_priv[0]** \_\_attribute\_\_ ((aligned(sizeof(s64)))); /\*具体主机控制器私有数据结构\*/

};

usb\_hcd结构体主要成员简介如下：

●**flags：**标记成员，取值定义在结构体内，如下所示：

#define HCD\_FLAG\_HW\_ACCESSIBLE 0 /\*主机控制器可访问（操作）\*/

#define HCD\_FLAG\_POLL\_RH 2 /\* poll for rh status? \*/

#define HCD\_FLAG\_POLL\_PENDING 3 /\*状态是否已经改变\*/

#define HCD\_FLAG\_WAKEUP\_PENDING 4 /\*根HUB正在唤醒吗? \*/

#define HCD\_FLAG\_RH\_RUNNING 5 /\*根HUB正在运行吗？\*/

#define HCD\_FLAG\_DEAD 6 /\* controller has died? \*/

/\*标记位检测宏如下\*/

#define HCD\_HW\_ACCESSIBLE(hcd) ((hcd)->flags & (1U << HCD\_FLAG\_HW\_ACCESSIBLE))

#define HCD\_POLL\_RH(hcd) ((hcd)->flags & (1U << HCD\_FLAG\_POLL\_RH))

#define HCD\_POLL\_PENDING(hcd) ((hcd)->flags & (1U << HCD\_FLAG\_POLL\_PENDING))

#define HCD\_WAKEUP\_PENDING(hcd) ((hcd)->flags & (1U << HCD\_FLAG\_WAKEUP\_PENDING))

#define HCD\_RH\_RUNNING(hcd) ((hcd)->flags & (1U << HCD\_FLAG\_RH\_RUNNING))

#define HCD\_DEAD(hcd) ((hcd)->flags & (1U << HCD\_FLAG\_DEAD))

●**state：**状态成员，取值定义在结构体内，如下：

#define \_\_ACTIVE 0x01

#define \_\_SUSPEND 0x04

#define \_\_TRANSIENT 0x80

#define HC\_STATE\_HALT 0

#define HC\_STATE\_RUNNING (\_\_ACTIVE)

#define HC\_STATE\_QUIESCING (\_\_SUSPEND|\_\_TRANSIENT|\_\_ACTIVE)

#define HC\_STATE\_RESUMING (\_\_SUSPEND|\_\_TRANSIENT)

#define HC\_STATE\_SUSPENDED (\_\_SUSPEND)

#define HC\_IS\_RUNNING(state) ((state) & \_\_ACTIVE)

#define HC\_IS\_SUSPENDED(state) ((state) & \_\_SUSPEND)

●**irq：**主机控制器中断编号。

●**high\_prio\_bh/low\_prio\_bh：**giveback\_urb\_bh结构体成员，定义在/include/linux/usb/hcd.h头文件：

struct giveback\_urb\_bh {

bool running;

spinlock\_t lock;

struct list\_head **head**; /\*管理urb实例\*/

struct tasklet\_struct **bh**; /\*tasklet实例\*/

struct usb\_host\_endpoint \*completing\_ep;

};

giveback\_urb\_bh结构体用于管理传输完成的urb实例，执行传输完成后的工作。

●**pool[]：**dma\_pool结构体指针数组，表示DMA缓存池，项数为4，结构体定义在/mm/dmapool.c文件内：

struct dma\_pool { /\* the pool \*/

struct list\_head **page\_list**;

spinlock\_t lock;

size\_t **size**; /\*4个数组项，size大小依次为32、128、512、2048\*/

struct device \*dev;

size\_t allocation;

size\_t boundary;

char name[32];

struct list\_head pools;

};

●**hcd\_priv[0]：**主机控制器私有数据结构，后面介绍EHCI主机控制器驱动时将会涉及到。

**●self：**usb\_bus结构体成员，见下文。

**●driver：**指向hc\_driver结构体，由具体主机控制器驱动程序实现，定义见下文。

##### ■usb\_bus

usb\_hcd结构体中self成员是usb\_bus结构体，定义如下（/include/linux/usb.h）：

struct usb\_bus {

struct device \***controller**; /\*指向表示主机控制器的device实例，嵌入到xxx\_device结构体中\*/

int **busnum**; /\*主机控制器编号\*/

const char \*bus\_name; /\* stable id (PCI slot\_name etc) \*/

u8 uses\_dma; /\*主机控制器是否使用DMA\*/

u8 uses\_pio\_for\_control;

u8 otg\_port; /\* 0, or number of OTG/HNP port \*/

unsigned is\_b\_host:1; /\* true during some HNP roleswitches \*/

unsigned b\_hnp\_enable:1; /\* OTG: did A-Host enable HNP? \*/

unsigned no\_stop\_on\_short:1;

unsigned no\_sg\_constraint:1; /\* no sg constraint \*/

unsigned sg\_tablesize; /\* 0 or largest number of sg list entries \*/

int devnum\_next;

struct usb\_devmap **devmap**; /\*分配总线下设备地址的位图，/include/linux/usb.h\*/

struct usb\_device \***root\_hub**; /\*指向表示根HUB的USB设备usb\_device实例\*/

struct usb\_bus \*hs\_companion; /\* Companion EHCI bus, if any \*/

struct list\_head **bus\_list**; /\*添加到usb\_bus\_list全局双链表\*/

struct mutex usb\_address0\_mutex; /\* unaddressed device mutex \*/

int bandwidth\_allocated;

int bandwidth\_int\_reqs; /\*number of Interrupt requests \*/

int bandwidth\_isoc\_reqs; /\*number of Isoc. requests \*/

unsigned resuming\_ports; /\*bit array: resuming root-hub ports \*/

#if defined(CONFIG\_USB\_MON) || defined(CONFIG\_USB\_MON\_MODULE)

struct mon\_bus \*mon\_bus; /\* non-null when associated \*/

int monitored; /\* non-zero when monitored \*/

#endif

};

##### ■hc\_driver

usb\_hcd结构体中driver成员是hc\_driver结构体指针。hc\_driver结构体是由具体USB主机控制器驱动程序实现的数据结构。

hc\_driver结构体定义如下（/include/linux/usb/hcd.h）：

struct hc\_driver {

const char \*description; /\*描述符，如"ehci-hcd"等\*/

const char \*product\_desc; /\*厂商/产品字符串\*/

size\_t **hcd\_priv\_size**; /\*usb\_hcd实例中私有数据大小，例如：ehci\_hcd\*/

irqreturn\_t (**\*irq**) (struct usb\_hcd \*hcd); /\*主机控制器中断处理函数\*/

int **flags**; /\*标记成员，取值见下文\*/

int (\***reset**) (struct usb\_hcd \*hcd); /\*初始化HCD或根HUB\*/

int (\***start**) (struct usb\_hcd \*hcd);

int (\*pci\_suspend)(struct usb\_hcd \*hcd, bool do\_wakeup);

int (\*pci\_resume)(struct usb\_hcd \*hcd, bool hibernated);

void (\*stop) (struct usb\_hcd \*hcd); /\*使HCD停止写内存和进行I/O操作\*/

void (\*shutdown) (struct usb\_hcd \*hcd); /\*关闭HCD\*/

int (\*get\_frame\_number) (struct usb\_hcd \*hcd); /\*返回当前帧编号\*/

/\*管理IO请求和设备状态（读写请求）\*/

int (\***urb\_enqueue**)(struct usb\_hcd \*hcd,struct urb \*urb, gfp\_t mem\_flags); /\*urb入队并执行传输\*/

int (\***urb\_dequeue**)(struct usb\_hcd \*hcd,struct urb \*urb, int status);

int (\*map\_urb\_for\_dma)(struct usb\_hcd \*hcd, struct urb \*urb,gfp\_t mem\_flags);

void (\*unmap\_urb\_for\_dma)(struct usb\_hcd \*hcd, struct urb \*urb);

void (\*endpoint\_disable)(struct usb\_hcd \*hcd,struct usb\_host\_endpoint \*ep); /\*释放端点资源\*/

void (\*endpoint\_reset)(struct usb\_hcd \*hcd,struct usb\_host\_endpoint \*ep); /\*复位端点\*/

/\*根HUB支持\*/

int (\*hub\_status\_data) (struct usb\_hcd \*hcd, char \*buf);

int (\***hub\_control**) (struct usb\_hcd \*hcd,u16 typeReq, u16 wValue, u16 wIndex, \

char \*buf, u16 wLength); /\*实现控制传输\*/

int (\*bus\_suspend)(struct usb\_hcd \*);

int (\*bus\_resume)(struct usb\_hcd \*);

int (\*start\_port\_reset)(struct usb\_hcd \*, unsigned port\_num);

void (\*relinquish\_port)(struct usb\_hcd \*, int);

int (\*port\_handed\_over)(struct usb\_hcd \*, int);

void (\*clear\_tt\_buffer\_complete)(struct usb\_hcd \*,struct usb\_host\_endpoint \*);

/\* xHCI主机控制器回调函数（USB3.0） \*/

...

};

hc\_driver结构体主要成员简介如下：

●**flags**：标记，主要表示主机控制器支持的USB规范版本，如下所示：

#define HCD\_MEMORY 0x0001 /\* HC regs use memory (else I/O) \*/

#define HCD\_LOCAL\_MEM 0x0002 /\*需要本地内存\*/

#define HCD\_SHARED 0x0004 /\*控制器共享硬件\*/

#define HCD\_USB11 0x0010 /\* USB 1.1 \*/

#define HCD\_USB2 0x0020 /\* USB 2.0 \*/

#define HCD\_USB25 0x0030 /\* Wireless USB 1.0 (USB 2.5)\*/

#define HCD\_USB3 0x0040 /\* USB 3.0 \*/

#define **HCD\_MASK**  0x0070 /\*规范版本的掩码\*/

#define HCD\_BH 0x0100 /\* URB complete in BH context \*/

●**hub\_control()：**实现控制传输的函数。

●**urb\_enqueue()/urb\_dequeue()：**其它传输类型时，将urb添加/移出队列。

#### 2主机控制器驱动

USB主机控制器作为设备，挂接在某个总线上，如platform总线、PCI总线等，因此需要定义并注册表示主机控制器的xxx\_device实例（设备）和表示主机控制器驱动的xxx\_driver实例。

主机控制器驱动程序需要实现hc\_driver实例，xxx\_driver实例中的probe()函数需要创建、设置并添加表示主机控制器的usb\_hcd实例，如下图所示。



USB驱动核心层提供了分配和添加主机控制器**usb\_hcd**实例的接口函数，供USB主机控制器驱动程序中的probe()函数调用。

##### ■创建usb\_hcd实例

创建主机控制器usb\_hcd实例的接口函数如下：

struct usb\_hcd \***usb\_create\_hcd**(const struct hc\_driver \*driver,struct device \*dev, const char \*bus\_name)；

参数语义如下：

driver：指向具体控制器驱动实现的hc\_driver实例。

dev：指向表示主机控制器设备的device实例，即xxx\_device结构体中内嵌的device实例。

bus\_name：指向控制器名称字符串。

usb\_create\_hcd()函数定义如下（/drivers/usb/core/hcd.c）：

struct usb\_hcd \***usb\_create\_hcd**(const struct hc\_driver \*driver,struct device \*dev, const char \*bus\_name)

{

return **usb\_create\_shared\_hcd**(driver, dev, bus\_name, NULL); /\*/drivers/usb/core/hcd.c\*/

}

usb\_create\_shared\_hcd()函数代码简列如下，用于创建和初始化usb\_hcd实例：

struct usb\_hcd \*usb\_create\_shared\_hcd(const struct hc\_driver \***driver**, \

struct device \*dev, const char \*bus\_name,struct usb\_hcd \*primary\_hcd)

/\*primary\_hcd：NULL\*/

{

struct usb\_hcd \*hcd;

**hcd = kzalloc(sizeof(\*hcd) + driver->hcd\_priv\_size, GFP\_KERNEL)**;

/\*分配usb\_hcd实例，后接私有数据\*/

...

if (primary\_hcd == NULL) {

hcd->bandwidth\_mutex = kmalloc(sizeof(\*hcd->bandwidth\_mutex),GFP\_KERNEL);

...

mutex\_init(hcd->bandwidth\_mutex);

**dev\_set\_drvdata(dev, hcd)**; /\***platform\_device->dev->driver\_data=hcd**\*/

} else {

...

}

kref\_init(&hcd->kref); /\*引用计数初始化\*/

**usb\_bus\_init(&hcd->self)**; /\*初始化usb\_bus结构体成员，/drivers/usb/core/hcd.c\*/

**hcd->self.controller = dev**; /\*指向表示主机控制器的device实例\*/

hcd->self.bus\_name = bus\_name;

hcd->self.uses\_dma = (dev->dma\_mask != NULL);

**init\_timer(&hcd->rh\_timer)**; /\*初始化轮询根HUB定时器\*/

hcd->rh\_timer.function = **rh\_timer\_func**; /\*定时器到期处理函数为**rh\_timer\_func()**\*/

hcd->rh\_timer.data = (unsigned long) **hcd**; /\*定时器数据指向usb\_hcd实例\*/

#ifdef CONFIG\_PM

INIT\_WORK(&hcd->wakeup\_work, hcd\_resume\_work);

#endif

**hcd->driver = driver**; /\*关联hc\_driver实例\*/

hcd->speed = **driver->flags & HCD\_MASK**; /\*标识速率\*/

hcd->product\_desc = (driver->product\_desc) ? driver->product\_desc :"USB Host Controller";

return hcd; /\*返回usb\_hcd实例指针\*/

}

usb\_create\_shared\_hcd()函数将分配usb\_hcd实例及其私有数据，初始化和设置usb\_hcd实例，关联参数传递的hc\_driver实例，初始化内嵌定时器rh\_timer，其到期执行函数为**rh\_timer\_func()**。表示主机控制器的xxx\_device->dev->driver\_data成员指向hcd，即usb\_hcd实例。

##### ■添加usb\_hcd实例

USB主机控制器驱动probe()函数在创建完usb\_hcd实例后，还需要对实例进行特定于控制器的设置，最后调用**usb\_add\_hcd()**函数向USB驱动核心层添加usb\_hcd实例，函数定义如下（/drivers/usb/core/hcd.c）：

int usb\_add\_hcd(struct usb\_hcd \*hcd,unsigned int irqnum, unsigned long irqflags)

/\*hcd：usb\_hcd实例指针，irqnum：中断编号，irqflags：中断标记\*/

{

int retval;

struct usb\_device \*rhdev; /\*表示根HUB的usb\_device\*/

if (IS\_ENABLED(CONFIG\_USB\_PHY) && !hcd->usb\_phy) { /\*OTG\*/

struct usb\_phy \*phy = usb\_get\_phy\_dev(hcd->self.controller, 0);

if (IS\_ERR(phy)) {

...

} else {

retval = usb\_phy\_init(phy);

...

hcd->usb\_phy = phy;

hcd->remove\_phy = 1;

}

}

if (IS\_ENABLED(CONFIG\_GENERIC\_PHY) && !hcd->phy) {

struct phy \*phy = phy\_get(hcd->self.controller, "usb");

if (IS\_ERR(phy)) {

...

} else {

retval = phy\_init(phy);

if (retval) {

phy\_put(phy);

goto err\_phy;

}

retval = phy\_power\_on(phy);

...

hcd->phy = phy;

hcd->remove\_phy = 1;

}

}

dev\_info(hcd->self.controller, "%s\n", hcd->product\_desc);

if (authorized\_default < 0 || authorized\_default > 1)

hcd->authorized\_default = hcd->wireless ? 0 : 1;

else

hcd->authorized\_default = authorized\_default;

set\_bit(**HCD\_FLAG\_HW\_ACCESSIBLE**, &hcd->flags); /\*设置标记位\*/

if ((retval = **hcd\_buffer\_create**(hcd)) != 0) {

/\*为pool[]指针数组创建dma\_pool实例，/drivers/usb/core/buffer.c\*/

...

}

if ((retval = **usb\_register\_bus(&hcd->self))** < 0) /\*注册usb\_bus实例，/drivers/usb/core/hcd.c\*/

goto err\_register\_bus; /\*分配编号，添加到全局双链表，执行usb\_notifier\_list通知链等\*/

rhdev = **usb\_alloc\_dev**(NULL, &hcd->self, 0); /\*/drivers/usb/core/usb.c\*/

/\*创建表示根HUB的**usb\_device**实例，总线为usb\_bus\_type，设备类型为usb\_device\_type\*/

...

mutex\_lock(&usb\_port\_peer\_mutex);

**hcd->self.root\_hub = rhdev**; /\*指向表示根HUB的usb\_device实例\*/

mutex\_unlock(&usb\_port\_peer\_mutex);

switch (hcd->speed) { /\*主机控制器速度模式\*/

case HCD\_USB11:

rhdev->speed = USB\_SPEED\_FULL; /\*全速\*/

break;

case HCD\_USB2:

rhdev->speed = USB\_SPEED\_HIGH; /\*高速\*/

break;

case HCD\_USB25:

rhdev->speed = USB\_SPEED\_WIRELESS; /\*无线\*/

break;

case HCD\_USB3:

rhdev->speed = USB\_SPEED\_SUPER; /\*超高速\*/

break;

default:

retval = -EINVAL;

goto err\_set\_rh\_speed;

}

device\_set\_wakeup\_capable(&rhdev->dev, 1);

/\*dev->power.can\_wakeup = 1等，/drivers/base/power/wakeup.c\*/

set\_bit(HCD\_FLAG\_RH\_RUNNING, &hcd->flags); /\*设置主机控制器标记位\*/

if (hcd->driver->reset && (retval = **hcd->driver->reset(hcd)**) < 0) { /\*初始化控制器和根HUB\*/

...

}

hcd->rh\_pollable = 1;

if (device\_can\_wakeup(hcd->self.controller)&& device\_can\_wakeup(&hcd->self.root\_hub->dev))

dev\_dbg(hcd->self.controller, "supports USB remote wakeup\n");

/\*初始化giveback\_urb\_bh结构体成员，tasklet执行函数为**usb\_giveback\_urb\_bh()**\*/

**init\_giveback\_urb\_bh(&hcd->high\_prio\_bh)**; /\*/drivers/usb/core/hcd.c\*/

**init\_giveback\_urb\_bh(&hcd->low\_prio\_bh)**;

if (usb\_hcd\_is\_primary\_hcd(hcd) && irqnum) {

retval = **usb\_hcd\_request\_irqs**(hcd, irqnum, irqflags); /\*/drivers/usb/core/hcd.c\*/

/\*申请中断，中断处理函数为usb\_hcd\_irq()，函数内调用**hcd->driver->irq(hcd)**函数\*/

...

}

hcd->state = HC\_STATE\_RUNNING;

retval = **hcd->driver->start(hcd)**; /\*启动主机控制器\*/

...

if ((retval = **register\_root\_hub(hcd)**) != 0)

/\*注册表示根HUB的usb\_device实例，将匹配通用USB设备驱动，/drivers/usb/core/hcd.c\*/

...

retval = sysfs\_create\_group(&rhdev->dev.kobj, &usb\_bus\_attr\_group); /\*添加属性文件\*/

...

if (hcd->uses\_new\_polling && HCD\_POLL\_RH(hcd))

usb\_hcd\_poll\_rh\_status(hcd); /\*激活轮询定时器，检测根HUB状态变化\*/

return retval;

...

}

usb\_add\_hcd()函数执行的主要工作简介如下：

（1）为usb\_hcd结构体pool[]指针数组分配dma\_pool实例。

（2）注册usb\_hcd内嵌的usb\_bus结构体实例，主要工作是为其分配编号，添加到全局双链表，执行usb\_notifier\_list通知链等。

（3）创建表示根HUB的usb\_device实例，挂载到usb\_bus\_type总线（USB总线），设备类型设为usb\_device\_type，并将实例赋予hcd->self.root\_hub。

（4）复位主机控制器和根HUB。

（5）初始化usb\_hcd中giveback\_urb\_bh结构体成员high\_prio\_bh和low\_prio\_bh，其内嵌tasklet\_struct的执行函数为usb\_giveback\_urb\_bh()，用于处理giveback\_urb\_bh中管理的传输完成的urb实例。

（6）申请主机控制器中断，中断处理函数为usb\_hcd\_irq()，函数内调用hcd->driver->irq(hcd)函数。

（7）注册根HUB，添加表示根HUB的usb\_device实例（将匹配通用USB设备驱动），添加属性文件等。

创建并注册usb\_hcd实例的结果简列如下图所示： 

USB主机控制器中内嵌一个根HUB，它被视为USB设备，在添加usb\_hcd实例时，将创建并注册表示根HUB的usb\_device实例。此usb\_hcd实例将匹配通用的USB设备驱动，其probe()函数将选择设备配置，获取接口信息，为接口创建并注册usb\_interface实例。usb\_interface实例又将匹配usb\_driver实例，加载接口驱动程序，详细内容见下一小节（上图中未画出）。

#### 3 EHCI控制器驱动

USB主机控制器驱动程序位于/drivers/usb/host/目录下。

EHCI主机控制器是支持USB2.0规范的标准USB主机控制器，内嵌OHCI或UHCI控制器，下面以EHCI主机控制器驱动为例简要说明主机控制器驱动的实现。

EHCI主机控制器包Capability、Operational和EHCI实现相关的寄存器，主机控制器的操作主要就是对其寄存器的操作，这里就不详细介绍了，请读者参考《Enhanced Host Controller Interface Specification for

Universal Serial Bus》手册。下面主要介绍内核中EHCI主机控制器驱动框架。

EHCI主机控制器驱动主要实现代码位于ehci-hcd.c、ehci-hub.c、ehci-sched.c、ehci-mem.c等文件内，主要是定义hc\_driver实例**ehci\_hc\_driver**及其操作函数等。

在ehci-xxx.c文件内，主要是定义并注册表示主机控制器驱动的xxx\_driver实例，实例prob()函数中将创建、设置并添加表示主机控制器的usb\_hcd实例。

例如，假设主机控制器挂接在平台总线上，在ehci-platform.c文件内定义并注册了表示主机控制器驱动的platform\_driver实例ehci\_platform\_driver，在其probe()函数中将创建并添加usb\_hcd实例，其关联的hc\_driver实例设为ehci\_platform\_hc\_driver，它是ehci\_hc\_driver实例的副本，只不过加上了私有数据。

下面以挂接在平台总线的EHCI主机控制器为例，简要说明其驱动实现。

##### ■设备

如果ECHI控制器挂接在platform总线上，则在/drivers/usb/host/ehci-platform.c文件内实现了通用的主机控制器驱动程序（需选择USB\_EHCI\_HCD\_PLATFORM配置选项）。

在板级相关文件（或设备树节点）中需要向驱动传递主机控制器的硬件信息，由usb\_ehci\_pdata结构体表示，定义如下（/include/linux/usb/ehci\_pdriver.h）：

struct usb\_ehci\_pdata {

int caps\_offset;

unsigned has\_tt:1;

unsigned has\_synopsys\_hc\_bug:1;

unsigned big\_endian\_desc:1;

unsigned big\_endian\_mmio:1;

unsigned no\_io\_watchdog:1;

unsigned reset\_on\_resume:1;

unsigned dma\_mask\_64:1;

int (\*power\_on)(struct platform\_device \*pdev); /\*开启所有供电及时钟\*/

void (\*power\_off)(struct platform\_device \*pdev); /\*关闭所有供电及时钟\*/

void (\*power\_suspend)(struct platform\_device \*pdev);

int (\*pre\_setup)(struct usb\_hcd \*hcd);

};

在龙芯1B板级文件中，定义了以下usb\_ehci\_pdata结构体实例：

static struct usb\_ehci\_pdata **ls1x\_ehci\_pdata** = { }; /\*usb\_ehci\_pdata实例，为空\*/

struct platform\_device **ls1x\_ehci\_pdev** = { /\*表示主机控制器设备的platform\_device实例\*/

.name = **"ehci-platform"**, /\*名称，用于匹配驱动\*/

.id = -1,

.num\_resources = ARRAY\_SIZE(ls1x\_ehci\_resources),

.resource = **ls1x\_ehci\_resources**, /\*资源\*/

.dev = {

.dma\_mask = &ls1x\_ehci\_dmamask,

.platform\_data = **&ls1x\_ehci\_pdata**, /\*指向usb\_ehci\_pdata实例\*/

},

};

ls1x\_ehci\_resources为资源数组定义如下：

static u64 ls1x\_ehci\_dmamask = DMA\_BIT\_MASK(32);

static struct resource ls1x\_ehci\_resources[] = { /\*主机控制器资源\*/

[0] = {

.start = LS1X\_EHCI\_BASE, /\*控制寄存器基址\*/

.end = LS1X\_EHCI\_BASE + SZ\_32K - 1,

.flags= IORESOURCE\_MEM, /\*内存\*/

},

[1] = {

.start = LS1X\_EHCI\_IRQ, /\*中断编号\*/

.flags= IORESOURCE\_IRQ, /\*中断\*/

},

};

在平台定义的初始化函数中将注册ls1x\_ehci\_pdev实例（platform\_device），代码简列如下：

static struct platform\_device \***ls1b\_platform\_devices**[] \_\_initdata = {

...

&ls1x\_ehci\_pdev,

...

};

static int \_\_init ls1b\_platform\_init(void)

{

int err;

ls1x\_serial\_setup(&ls1x\_uart\_pdev);

err = platform\_add\_devices(**ls1b\_platform\_devices**,ARRAY\_SIZE(ls1b\_platform\_devices));

return err;

}

arch\_initcall(ls1b\_platform\_init);

##### ■驱动

在/drivers/usb/host/ehci-platform.c文件内定义了匹配平台EHCI主机控制器设备的驱动platform\_driver实例，如下：

static struct platform\_driver ehci\_platform\_driver = {

.id\_table = ehci\_platform\_table,

.probe = **ehci\_platform\_probe**, /\*驱动探测函数\*/

.remove = ehci\_platform\_remove,

.shutdown = usb\_hcd\_platform\_shutdown,

.driver = {

.name = **"ehci-platform"**, /\*名称，用于匹配设备\*/

.pm = &ehci\_platform\_pm\_ops,

.of\_match\_table = vt8500\_ehci\_ids,

}

};

在驱动探测函数ehci\_platform\_probe()中将创建、设置和添加表示主机控制器的usb\_hcd实例，执行结果如下图所示，源代码请读者自行阅读：



usb\_hcd关联的hc\_driver实例为**ehci\_platform\_hc\_driver**，它复制于通用ehci\_hc\_driver实例。usb\_hcd后面的私有数据结构为ehci\_hcd，其最后一个成员是platform总线EHCI主机控制器的ehci\_platform\_priv私有数据结构。

ehci\_hcd结构体定义在/drivers/usb/host/ehci.h头文件。

ehci\_platform\_priv结构体定义在/drivers/usb/host/ehci-platform.c文件内。

内核在/drivers/usb/host/ehci-hcd.c文件内定义了ECHI主机控制器通用的hc\_driver实例ehci\_hc\_driver，如下所示：

static const struct hc\_driver **ehci\_hc\_driver** = {

.description = hcd\_name,

.product\_desc = "EHCI Host Controller",

.hcd\_priv\_size = sizeof(struct **ehci\_hcd**), /\*私有数据\*/

.irq = **ehci\_irq**, /\*中断处理函数\*/

.flags = HCD\_MEMORY | HCD\_USB2 | HCD\_BH,

.reset = ehci\_setup,

.start = ehci\_run,

.stop = ehci\_stop,

.shutdown = ehci\_shutdown,

.urb\_enqueue = **ehci\_urb\_enqueue**, /\*urb入队，还要调度urb的传输\*/

.urb\_dequeue = ehci\_urb\_dequeue,

.endpoint\_disable = ehci\_endpoint\_disable,

.endpoint\_reset = ehci\_endpoint\_reset,

.clear\_tt\_buffer\_complete = ehci\_clear\_tt\_buffer\_complete,

.get\_frame\_number = ehci\_get\_frame,

.hub\_status\_data = ehci\_hub\_status\_data, /\*中断轮询定时器中调用\*/

.hub\_control = **ehci\_hub\_control**, /\*实现控制传输\*/

.bus\_suspend = ehci\_bus\_suspend,

.bus\_resume = ehci\_bus\_resume,

.relinquish\_port = ehci\_relinquish\_port,

.port\_handed\_over = ehci\_port\_handed\_over,

.free\_dev = ehci\_remove\_device,

};

ehci\_hc\_driver实例中操作函数定义在ehci-hcd.c、ehci-hub.c、ehci-sched.c、ehci-mem.c等文件内，请读者自行阅读源代码。

在/drivers/usb/host/ehci-platform.c文件内定义了以下初始化（模块加载）函数：

static int \_\_init ehci\_platform\_init(void)

{

if (usb\_disabled())

return -ENODEV;

pr\_info("%s: " DRIVER\_DESC "\n", hcd\_name);

**ehci\_init\_driver**(&ehci\_platform\_hc\_driver, &platform\_overrides);

/\*由ehci\_hc\_driver生成ehci\_platform\_hc\_driver实例\*/

return **platform\_driver\_register**(&ehci\_platform\_driver); /\*注册ehci\_platform\_driver实例\*/

}

module\_init(ehci\_platform\_init); /\*模块加载（初始化）函数\*/

ehci\_init\_driver()用于复制ehci\_hc\_driver实例数据至ehci\_platform\_hc\_driver实例，并修改私有数据大小等部分成员。ehci\_platform\_init()函数最后注册ehci\_platform\_driver实例，与ls1x\_ehci\_pdev实例匹配后，将调用probe()函数，创建、添加表示EHCI主机控制器的usb\_hcd实例及其私有数据。

### 8.9.4设备与驱动

内核在/drivers/usb/core/driver.c文件内定义了USB总线实例：

struct bus\_type usb\_bus\_type = {

.name = "usb",

.match = **usb\_device\_match**, /\*匹配函数，后面再介绍\*/

.uevent = usb\_uevent,

};

USB总线设备链表下挂载的有设备usb\_device实例、接口usb\_interface实例等，驱动链表下挂载有设备驱动usb\_device\_driver实例、接口驱动usb\_driver实例等。

USB设备usb\_device实例和接口usb\_interface实例都是在探测设备时，由USB驱动核心层自动创建的。USB设备匹配的通用设备驱动usb\_generic\_driver实例也由内核定义和注册，因此实际需要用户实现和注册的就是接口驱动usb\_driver实例。

#### 1设备

USB设备是表现出一种或多种功能的逻辑或物理实体。HUB与周边设备都属于设备。主机会为每个总线上的设备指定一个专属地址。复合设备含有HUB以及一个或多个永久相连的设备。主机以（与对待单一设备）相同的方式对待复合设备，就如同这个集线器和其功能是单独的物理设备一样。每个HUB和嵌入式设备都拥有独特的地址。

复合设备只有1个总线地址，但有多个独立接口，每个接口提供一种功能。例如，复合设备可同时含有用于大容量存储设备和键盘的接口。不同接口可使用主机上不同的驱动程序，通常说的USB设备驱动程序是匹配USB设备接口的，而不是USB设备。

USB驱动框架中设备由usb\_device结构体表示，接口由usb\_interface结构体表示。USB设备下接口usb\_interface实例由设备usb\_device实例管理。

##### ■usb\_device

USB设备在内核中由usb\_device结构体表示，定义如下（/include/linux/usb.h）：

struct usb\_device {

int **devnum**; /\*设备编号，设备地址\*/

char devpath[16]; /\*设备ID\*/

u32 route; /\*xHCI（USB3.0）\*/

enum usb\_device\_state **state**; /\*设备状态：配置、地址等\*/

enum usb\_device\_speed **speed**; /\*速度模式，高速、全速、低速\*/

struct usb\_tt \*tt; /\*事务转换器，适用于高速HUB\*/

int ttport; /\*事务转换器端口数\*/

unsigned int toggle[2]; /\*位图，标记端点是输入（0）还是输出（1）\*/

struct usb\_device \*parent; /\*父设备，即集线器\*/

struct usb\_bus \***bus**; /\*主机控制器结构中嵌入的usb\_bus结构体实例\*/

struct usb\_host\_endpoint **ep0**; /\*端点0（控制端点）信息（描述字），/include/linux/usb.h\*/

struct device **dev**; /\*内嵌的device实例\*/

struct usb\_device\_descriptor **descriptor**; /\*设备描述符，/include/uapi/linux/usb/ch9.h\*/

struct usb\_host\_bos \*bos; /\*二进制设备目标描述字，/include/linux/usb.h\*/

struct usb\_host\_config \***config**; /\*所有的设备配置（含配置描述字和接口），/include/linux/usb.h\*/

struct usb\_host\_config \***actconfig**; /\*当前活跃的设备配置\*/

struct usb\_host\_endpoint \***ep\_in[16]**; /\*输入端点列表，/include/linux/usb.h\*/

struct usb\_host\_endpoint \***ep\_out[16]**; /\*输出端点列表，/include/linux/usb.h\*/

char \*\*rawdescriptors; /\*各配置的原始描述字\*/

unsigned short bus\_mA; /\*从总线上可取的电流mA数\*/

u8 portnum; /\*父接口编号，初始为1\*/

u8 level; /\*在USB总线拓扑结构中的层级数\*/

unsigned can\_submit:1; /\*可以提交URB\*/

unsigned persist\_enabled:1; /\*USB\_PERSIST使能\*/

unsigned have\_langid:1;

unsigned authorized:1;

unsigned authenticated:1;

unsigned wusb:1; /\*无线USB\*/

unsigned lpm\_capable:1;

unsigned usb2\_hw\_lpm\_capable:1;

unsigned usb2\_hw\_lpm\_besl\_capable:1;

unsigned usb2\_hw\_lpm\_enabled:1;

unsigned usb2\_hw\_lpm\_allowed:1;

unsigned usb3\_lpm\_enabled:1;

int string\_langid;

/\*静态字符串，厂商信息，序列号等\*/

char \*product;

char \*manufacturer;

char \*serial;

struct list\_head filelist; /\*usbfs\*/

int maxchild; /\*如果是HUB，表示端口数量\*/

u32 quirks;

atomic\_t urbnum; /\*提交URB的数量\*/

unsigned long active\_duration; /\*设备活跃时长\*/

#ifdef CONFIG\_PM

unsigned long connect\_time;

unsigned do\_remote\_wakeup:1;

unsigned reset\_resume:1;

unsigned port\_is\_suspended:1;

#endif

struct wusb\_dev \*wusb\_dev; /\*无线USB\*/

int slot\_id;

enum usb\_device\_removable removable;

struct usb2\_lpm\_parameters l1\_params;

struct usb3\_lpm\_parameters u1\_params;

struct usb3\_lpm\_parameters u2\_params;

unsigned lpm\_disable\_count;

};

usb\_device结构体主要成员简介如下：

●**devnum：**设备地址。

●**state：**设备状态，由usb\_device\_state枚举类型表示，定义如下（/include/uapi/linux/usb/ch9.h）：

enum usb\_device\_state {

USB\_STATE\_NOTATTACHED = 0,

USB\_STATE\_ATTACHED,

USB\_STATE\_POWERED, /\* wired \*/

USB\_STATE\_RECONNECTING, /\* auth \*/

USB\_STATE\_UNAUTHENTICATED, /\* auth \*/

USB\_STATE\_DEFAULT, /\*默认\*/

USB\_STATE\_ADDRESS, /\*地址\*/

USB\_STATE\_CONFIGURED, /\*配置\*/

USB\_STATE\_SUSPENDED

};

●**speed**：速度模式，由usb\_device\_speed枚举类型表示，定义如下（/include/uapi/linux/usb/ch9.h）：

enum usb\_device\_speed {

USB\_SPEED\_UNKNOWN = 0, /\* enumerating \*/

USB\_SPEED\_LOW, USB\_SPEED\_FULL, /\* usb 1.1 \*/

USB\_SPEED\_HIGH, /\* usb 2.0 \*/

USB\_SPEED\_WIRELESS, /\* wireless (usb 2.5) \*/

USB\_SPEED\_SUPER, /\* usb 3.0 \*/

};

●**dev**：通用设备device实例，挂接到USB总线usb\_bus\_type。

usb\_device结构体中包含设备的各类描述字信息，各描述字数据结构定义在/include/uapi/linux/usb/ch9.h头文件，这里就不一一列出各数据结构的定义了，只用图示的方式示意一下usb\_device结构体中包含的描述字信息。



在上图中指针数组只画出了一个指针项，例如ep\_in[]是一个16项的指针数组，数组项指向包含端点描述字的usb\_host\_endpoint结构体实例。

##### ■usb\_interface

usb\_interface结构体表示USB设备接口，定义在/include/linux/usb.h头文件：

struct usb\_interface {

struct usb\_host\_interface \***altsetting**; /\*指向替代设置usb\_host\_interface实例列表\*/

struct usb\_host\_interface \*cur\_altsetting; /\*当前激活的替代设置\*/

unsigned num\_altsetting; /\*替代设置数量\*/

struct usb\_interface\_assoc\_descriptor \*intf\_assoc;

int minor; /\*接口绑定的从设备号，如果使用USB设备主设备号有效\*/

enum usb\_interface\_condition condition; /\*接口绑定状态\*/

unsigned sysfs\_files\_created:1; /\* the sysfs attributes exist \*/

unsigned ep\_devs\_created:1; /\* endpoint "devices" exist \*/

unsigned unregistering:1; /\*接口未注册时设置\*/

unsigned needs\_remote\_wakeup:1; /\*设置表示驱动要求具有远程唤醒能力\*/

unsigned needs\_altsetting0:1; /\* switch to altsetting 0 is pending \*/

unsigned needs\_binding:1; /\* needs delayed unbind/rebind \*/

unsigned resetting\_device:1; /\* true: bandwidth alloc after reset \*/

struct device **dev**; /\*表示接口的device实例\*/

struct device **\***usb\_dev; /\*\*/

atomic\_t pm\_usage\_cnt; /\*电源管理使用计数\*/

struct work\_struct reset\_ws; /\* for resets in atomic context \*/

};

在USB设备插入到HUB端口时，HUB端口状态变化后，主机控制器将会检测到，并动态创建、设置和添加表示USB设备usb\_device实例。

创建usb\_device实例的函数为usb\_alloc\_dev()（/drivers/usb/core/usb.c），添加usb\_device实例函数为usb\_new\_device()（/drivers/usb/core/hub.c）。添加设备时会选择设备配置，获取接口信息，创建并注册接口usb\_interface实例（后面将介绍函数实现）。

#### 2驱动

USB设备和接口匹配的驱动分别为设备驱动usb\_device\_driver和接口驱动usb\_driver。

##### ■设备驱动

USB设备驱动由usb\_device\_driver结构体表示，定义如下（/include/linux/usb.h）：

struct usb\_device\_driver {

const char \*name; /\*名称\*/

int (\***probe**) (struct usb\_device \*udev); /\*探测函数\*/

void (\*disconnect) (struct usb\_device \*udev); /\*断开连接\*/

int (\*suspend) (struct usb\_device \*udev, pm\_message\_t message);

int (\*resume) (struct usb\_device \*udev, pm\_message\_t message);

struct usbdrv\_wrap **drvwrap**; /\*内嵌驱动device\_driver实例\*/

unsigned int supports\_autosuspend:1;

};

usb\_device\_driver结构体中drvwrap成员为usbdrv\_wrap结构体，定义如下（/include/linux/usb.h）：

struct usbdrv\_wrap {

struct device\_driver **driver**;

int for\_devices; /\*0表示接口驱动，非0表示设备驱动\*/

};

注册usb\_device\_driver实例的函数声明如下（定义在/drivers/usb/core/driver.c）：

int **usb\_register\_device\_driver**(struct usb\_device\_driver \*,struct module \*); /\*/include/linux/usb.h\*/

void **usb\_register(driver)**; /\*当前模块\*/

以上两个函数比较简单，主要是注册usb\_device\_driver.drvwrap.driver表示的device\_driver实例，挂接到USB总线usb\_bus\_type，usb\_device\_driver.drvwrap.driver.probe()函数设为**usb\_probe\_device()**，函数内调用usb\_device\_driver.probe()函数。

注意：设备驱动usb\_device\_driver实例，可匹配所有的usb\_device实例。

注销usb\_device\_driver实例的函数声明如下（/include/linux/usb.h）：

void usb\_deregister\_device\_driver(struct usb\_device\_driver \*);

内核在/drivers/usb/core/generic.c定义了usb\_device\_driver实例**usb\_generic\_driver**，它匹配所有的设备，在其prob()函数将选择设备配置，获取配置下的接口描述字，为接口创建并注册usb\_interface实例等。

##### ■接口驱动

USB接口驱动由usb\_driver结构体表示，这才是我们通常在驱动程序中说的USB设备驱动，结构体定义如下（/include/linux/usb.h）：

struct usb\_driver {

const char \***name**; /\*名称\*/

int (\***probe**) (struct usb\_interface \*intf,const struct usb\_device\_id \*id); /\*探测函数，必须\*/

void (\***disconnect**) (struct usb\_interface \*intf); /\*断开接口函数，必须\*/

int (\*unlocked\_ioctl) (struct usb\_interface \*intf, unsigned int code,void \*buf); /\*设备控制\*/

int (\*suspend) (struct usb\_interface \*intf, pm\_message\_t message); /\*睡眠\*/

int (\*resume) (struct usb\_interface \*intf); /\*恢复\*/

int (\*reset\_resume)(struct usb\_interface \*intf); /\*复位\*/

int (\*pre\_reset)(struct usb\_interface \*intf); /\*由usb\_reset\_device()函数在设备复位前调用\*/

int (\*post\_reset)(struct usb\_interface \*intf); /\*由usb\_reset\_device()函数在设备复位后调用\*/

const struct usb\_device\_id \***id\_table**; /\*比对列表（标识列表），必须定义\*/

struct usb\_dynids **dynids**; /\*动态usb\_device\_id实例列表\*/

struct usbdrv\_wrap **drvwrap**; /\*内嵌device\_driver实例，for\_devices为0，见上文\*/

unsigned int no\_dynamic\_id:1;

unsigned int supports\_autosuspend:1; /\*1表示允许自动睡眠，0表示不允许自动睡眠\*/

unsigned int disable\_hub\_initiated\_lpm:1;

unsigned int soft\_unbind:1;

};

usb\_driver结构体主要成员简介如下：

●**probe()**：探测函数，匹配接口时调用，必须定义。

●**disconnect()：**断开接口函数，必须定义。

●**id\_table：**指向usb\_device\_id结构体数组（必须定义），用于匹配接口，结构体定义如下：

struct usb\_device\_id { /\*/include/linux/mod\_devicetable.h\*/

\_\_u16 match\_flags; /\*匹配标记，匹配下面哪类信息\*/

/\*厂商（设备）信息匹配\*/

\_\_u16 idVendor;

\_\_u16 idProduct;

\_\_u16 bcdDevice\_lo;

\_\_u16 bcdDevice\_hi;

/\*设备类信息匹配\*/

\_\_u8 bDeviceClass;

\_\_u8 bDeviceSubClass;

\_\_u8 bDeviceProtocol;

/\*接口类匹配\*/

\_\_u8 bInterfaceClass;

\_\_u8 bInterfaceSubClass;

\_\_u8 bInterfaceProtocol;

/\* Used for vendor-specific interface matches \*/

\_\_u8 bInterfaceNumber;

/\* not matched against \*/

kernel\_ulong\_t driver\_info

\_\_attribute\_\_((aligned(sizeof(kernel\_ulong\_t))));

};

在/include/linux/usb.h头文件中定义了匹配标记match\_flags的取值，以及初始化usb\_device\_id实例的宏，例如：

#define USB\_DEVICE\_ID\_MATCH\_DEVICE \ /\*匹配厂商、设备信息\*/

(USB\_DEVICE\_ID\_MATCH\_VENDOR | USB\_DEVICE\_ID\_MATCH\_PRODUCT)

...

#define USB\_DEVICE(vend, prod) \

.match\_flags = USB\_DEVICE\_ID\_MATCH\_DEVICE, \

.idVendor = (vend), \

.idProduct = (prod)

**●dynids：**usb\_dynids结构体成员，定义如下（/include/linux/usb.h）：

struct usb\_dynids {

spinlock\_t lock;

struct list\_head **list**; /\*双链表，管理动态usb\_device\_id实例\*/

};

usb\_dynids结构体中list双链表管理的是usb\_dynid结构体实例，定义如下（/include/linux/usb.h）：

struct usb\_dynid {

struct list\_head node;

**struct usb\_device\_id id**; /\*动态usb\_device\_id实例\*/

};

在USB总线匹配函数中，接口与接口驱动检查是否匹配时，先将接口信息（描述字）与id\_table指向usb\_device\_id实例列表中各项匹配，如果不匹配再与dynids链表管理的动态usb\_device\_id实例逐个匹配，以上有一个匹配成功，匹配函数返回1，否则返回0。

###### ●注册接口驱动

注册USB接口驱动的接口函数声明如下（/include/linux/usb.h）：

int **usb\_register\_driver**(struct usb\_driver \*, struct module \*,const char \*);

#define **usb\_register**(driver) \

usb\_register\_driver(driver, THIS\_MODULE, KBUILD\_MODNAME)

#define **module\_usb\_driver**(\_\_usb\_driver) \ /\*在模块中定义初始化函数，见下节\*/

module\_driver(\_\_usb\_driver, usb\_register, usb\_deregister)

void usb\_deregister(struct usb\_driver \*)：注销usb\_driver实例。

以上注册函数最终都是调用usb\_register\_driver()函数，定义如下（/drivers/usb/core/driver.c）：

int usb\_register\_driver(struct usb\_driver \*new\_driver, struct module \*owner,const char \*mod\_name)

/\*owner：模块指针，mod\_name：模块名称\*/

{

int retval = 0;

if (usb\_disabled()) /\*受nousb变量控制，默认为0，/drivers/usb/core/usb.c\*/

return -ENODEV;

new\_driver->drvwrap.for\_devices = **0**; /\*匹配接口，如果为1表示匹配USB设备\*/

new\_driver->drvwrap.driver.name = new\_driver->name; /\*驱动名称\*/

new\_driver->drvwrap.driver.bus = &**usb\_bus\_type**; /\*USB总线实例\*/

new\_driver->drvwrap.driver.probe = **usb\_probe\_interface**; /\*调用usb\_driver->probe()函数\*/

new\_driver->drvwrap.driver.remove = usb\_unbind\_interface;

new\_driver->drvwrap.driver.owner = owner;

new\_driver->drvwrap.driver.mod\_name = mod\_name; /\*模块名称\*/

spin\_lock\_init(&new\_driver->dynids.lock);

INIT\_LIST\_HEAD(&new\_driver->dynids.list);

retval = **driver\_register**(&new\_driver->drvwrap.driver); /\*注册驱动\*/

...

retval = **usb\_create\_newid\_files**(new\_driver); /\*在驱动对应的sysfs目录项下添加属性文件\*/

...

out:

return retval;

...

}

#### 3总线匹配函数

USB设备、接口都挂载到USB总线上的设备链表，USB设备驱动、接口驱动都挂到USB总线的驱动链表。设备驱动只能匹配设备，不能匹配接口，接口驱动只能匹配接口不能匹配设备。

USB总线匹配函数定义如下（/drivers/usb/core/driver.c）：

static int **usb\_device\_match**(struct device \*dev, struct device\_driver \*drv)

{

if (is\_usb\_device(dev)) { /\*USB设备，dev->type == &usb\_device\_type，/drivers/usb/core/usb.h\*/

if (!**is\_usb\_device\_driver**(drv)) /\*只有USB设备驱动才能匹配USB设备\*/

return 0;

return 1; /\*USB设备与任意usb\_device\_driver实例匹配\*/

} else if (**is\_usb\_interface(dev)**) { /\*接口，dev->type == &usb\_if\_device\_type\*/

struct usb\_interface \*intf;

struct usb\_driver \*usb\_drv;

const struct usb\_device\_id \*id;

if (is\_usb\_device\_driver(drv)) /\*只有usb\_driver才能与接口匹配\*/

return 0;

**intf = to\_usb\_interface(dev)**; /\*usb\_interface\*/

**usb\_drv = to\_usb\_driver(drv)**; /\*usb\_driver\*/

id = **usb\_match\_id(intf, usb\_drv->id\_table)**; /\*标识列表匹配，/drivers/usb/core/driver.c\*/

if (id)

return 1;

id = **usb\_match\_dynamic\_id(intf, usb\_drv)**; /\*/drivers/usb/core/driver.c\*/

if (id)

return 1;

}

return 0;

}

usb\_device\_match()函数只为表示USB设备和接口的device实例检查是否与驱动匹配，匹配流程如下：

（1）如果是USB设备usb\_device实例且驱动是usb\_device\_driver实例则匹配，任意usb\_device\_driver实例可与任意usb\_device实例匹配。usb\_device与所有的接口驱动usb\_driver都不匹配。

（2）如果是USB接口usb\_interface实例且驱动是usb\_driver实例，则调用usb\_match\_id()函数检查接口/设备中描述字信息，看其是否与usb\_driver->id\_table列表中某个列表项匹配（源代码请读者自行阅读），若匹配则函数返回1，否则进行下一步。

（3）对于USB接口，如果usb\_match\_id()函数检查不匹配，再调用usb\_match\_dynamic\_id()函数将接口/设备描述字中信息与usb\_driver->dynids.list动态usb\_device\_id实例双链表中实例匹配，若匹配则返回1，否则返回0。

#### 4初始化

USB总线驱动初始化函数usb\_init()定义在/drivers/usb/core/usb.c文件内，代码如下：

static int \_\_init usb\_init(void)

{

int retval;

if (usb\_disabled()) {

...

}

usb\_init\_pool\_max(); /\*初始化pool\_max[0]数组项，/drivers/usb/core/buffer.c\*/

retval = usb\_debugfs\_init();

/\*在debugfs中创建"usb"目录项和"devices"文件，/drivers/usb/core/usb.c\*/

...

usb\_acpi\_register(); /\*注册acpi\_bus\_type结构体usb\_acpi\_bus实例，/drivers/usb/core/usb-acpi.c\*/

retval = **bus\_register(&usb\_bus\_type)**; /\*注册USB总线实例\*/

...

retval = bus\_register\_notifier(&usb\_bus\_type, &usb\_bus\_nb); /\*总线通知链上添加usb\_bus\_nb通知\*/

...

retval = **usb\_major\_init()**; /\*注册cdev实例，主设备号为USB\_MAJOR，/drivers/usb/core/file.c\*/

...

retval = usb\_register(&usbfs\_driver); /\*注册usb\_driver实例usbfs\_driver，/drivers/usb/core/devio.c\*/

...

retval = usb\_devio\_init();

/\*注册cdev实例，主设备号为USB\_DEVICE\_MAJOR，/drivers/usb/core/devio.c\*/

...

retval = **usb\_hub\_init()**; /\*HUB驱动初始化（接口驱动），/drivers/usb/core/hub.c\*/

/\*注册usb\_driver实例hub\_driver，创建HUB工作队列等，见本节下文\*/

...

retval = usb\_register\_device\_driver(&**usb\_generic\_driver**, THIS\_MODULE);

/\*注册usb\_device\_driver实例usb\_generic\_driver，匹配所有usb\_device实例\*/

/\*/drivers/usb/core/generic.c\*/

...

return retval;

}

**subsys\_initcall(usb\_init)**; /\*内核启动阶段调用此函数\*/

### 8.9.5 HUB

USB主机控制器中内嵌根HUB，它被当作USB设备。在前面介绍的添加主机控制器usb\_hcd实例时，将为主机控制器中根HUB创建表示USB设备的usb\_device实例并注册。

注册usb\_device实例时，将匹配通用的USB设备驱动usb\_generic\_driver实例，在其probe()函数将为根HUB设备选择配置，获取接口信息，创建并注册接口usb\_interface实例。usb\_interface实例将匹配HUB接口驱动的usb\_driver结构体实例**hub\_driver**，其probe()函数将创建并注册表示HUB的usb\_hub实例等，如下图所示。



#### 1 HUB设备

在添加主机控制器usb\_hcd实例的usb\_add\_hcd()函数中，将创建和注册表示根HUB的usb\_device实例，函数调用关系如下：



##### ■创建USB设备

创建usb\_device实例的**usb\_alloc\_dev**()函数定义如下（/drivers/usb/core/usb.c）：

struct usb\_device \*usb\_alloc\_dev(struct usb\_device \*parent,

struct usb\_bus \*bus, unsigned port1)

/\*parent：指向连接到的HUB的usb\_device实例，此处为NULL，bus：hcd->self，port1：此处为0\*/

{

struct usb\_device \*dev;

struct usb\_hcd \*usb\_hcd = bus\_to\_hcd(bus);

unsigned root\_hub = 0;

**dev = kzalloc(sizeof(\*dev), GFP\_KERNEL)**; /\*分配usb\_device实例\*/

...

if (!usb\_get\_hcd(usb\_hcd)) { /\*引用计数加1\*/

...

}

/\* Root hubs aren't true devices, so don't allocate HCD resources ，根HUB不需要分配HCD资源\*/

if (usb\_hcd->driver->alloc\_dev && parent &&!usb\_hcd->driver->alloc\_dev(usb\_hcd, dev)) {

...

}

device\_initialize(&dev->dev); /\*初始化设备\*/

dev->dev.bus = &**usb\_bus\_type**; /\*总线\*/

dev->dev.type = &**usb\_device\_type**; /\*设备类型\*/

dev->dev.groups = usb\_device\_groups; /\*属性组\*/

dev->dev.dma\_mask = bus->controller->dma\_mask;

set\_dev\_node(&dev->dev, dev\_to\_node(bus->controller));

dev->state = USB\_STATE\_ATTACHED;

dev->lpm\_disable\_count = 1;

atomic\_set(&dev->urbnum, 0);

INIT\_LIST\_HEAD(&dev->ep0.urb\_list);

dev->ep0.desc.bLength = USB\_DT\_ENDPOINT\_SIZE; /\*端点0描述字长度\*/

dev->ep0.desc.bDescriptorType = USB\_DT\_ENDPOINT; /\*端点0描述字类型\*/

**usb\_enable\_endpoint(dev, &dev->ep0, false);**

/\*dev->ep0关联到dev->ep\_out[0]，端点0为输出，/drivers/usb/core/message.c\*/

dev->can\_submit = 1;

if (unlikely(!parent)) { /\*创建根HUB设备，没有父设备\*/

dev->devpath[0] = '0';

dev->route = 0;

dev->dev.parent = bus->controller; /\*表示主机控制器设备的device实例，嵌入到xxx\_device\*/

dev\_set\_name(&dev->dev, "usb%d", bus->busnum);

root\_hub = 1; /\*根HUB\*/

} else { /\*普通设备\*/

...

}

dev->portnum = port1; /\*根HUB为0\*/

dev->bus = bus; /\*usb\_hcd.usb\_bus\*/

**dev->parent = parent;** /\*根HUB为NULL\*/

INIT\_LIST\_HEAD(&dev->filelist);

#ifdef CONFIG\_PM

pm\_runtime\_set\_autosuspend\_delay(&dev->dev,usb\_autosuspend\_delay \* 1000);

dev->connect\_time = jiffies;

dev->active\_duration = -jiffies;

#endif

if (root\_hub) /\*根HUB \*/

dev->authorized = 1;

else {

dev->authorized = usb\_hcd->authorized\_default;

dev->wusb = usb\_bus\_is\_wusb(bus) ? 1 : 0;

}

return dev; /\*返回usb\_device实例指针\*/

}

##### ■注册根HUB

在添加usb\_hcd实例的usb\_add\_hcd()函数中将随后会调用**register\_root\_hub()**函数注册表示根HUB设备的usb\_device实例，函数定义如下（/drivers/usb/core/hcd.c）：

static int register\_root\_hub(struct usb\_hcd \*hcd)

{

struct device \***parent\_dev = hcd->self.controller**; /\*表示主机控制器的device实例\*/

struct usb\_device \***usb\_dev = hcd->self.root\_hub**; /\*表示根HUB的usb\_device实例\*/

const int devnum = 1; /\*根HUB设备地址为1\*/

int retval;

usb\_dev->devnum = devnum;

usb\_dev->bus->devnum\_next = devnum + 1;

memset (&usb\_dev->bus->devmap.devicemap, 0,sizeof usb\_dev->bus->devmap.devicemap);

/\*分配设备地址的位图清零\*/

set\_bit (devnum, usb\_dev->bus->devmap.devicemap); /\*设置根HUB地址1在位图中相应位\*/

usb\_set\_device\_state(usb\_dev, USB\_STATE\_ADDRESS); /\*设置设备状态\*/

mutex\_lock(&usb\_bus\_list\_lock);

usb\_dev->ep0.desc.wMaxPacketSize = cpu\_to\_le16(64);

/\*端点0在一个事务中可传输的最大数据长度（64字节）\*/

retval = **usb\_get\_device\_descriptor**(usb\_dev, USB\_DT\_DEVICE\_SIZE);

/\*读取设备描述字，写入usb\_dev.**descriptor**成员，/drivers/usb/core/message.c\*/

... /\*错误处理\*/

if (le16\_to\_cpu(usb\_dev->descriptor.bcdUSB) >= 0x0201) {

/\*设备描述字中BCD形式的设备版本号\*/

retval = usb\_get\_bos\_descriptor(usb\_dev);

if (!retval) {

usb\_dev->lpm\_capable = usb\_device\_supports\_lpm(usb\_dev);

} else if (usb\_dev->speed == USB\_SPEED\_SUPER) {

mutex\_unlock(&usb\_bus\_list\_lock);

dev\_dbg(parent\_dev, "can't read %s bos descriptor %d\n",dev\_name(&usb\_dev->dev), retval);

return retval;

}

}

retval = **usb\_new\_device (usb\_dev)**; /\*注册usb\_device实例，/drivers/usb/core/hub.c\*/

if (retval) {

... /\*注册失败\*/

} else { /\*注册成功\*/

spin\_lock\_irq (&hcd\_root\_hub\_lock);

**hcd->rh\_registered = 1**;

spin\_unlock\_irq (&hcd\_root\_hub\_lock);

if (HCD\_DEAD(hcd))

usb\_hc\_died (hcd); /\* This time clean up \*/

}

mutex\_unlock(&usb\_bus\_list\_lock);

return retval; /\*成功返回0\*/

}

usb\_device实例ep0.desc成员保存的是端点0描述字，descriptor成员保存的是设备描述字。

register\_root\_hub()函数主要工作如下：

（1）将根HUB的设备地址设为1，并设置地址位图中的位。

（2）设置ep0.desc成员，即端点0描述字。

（3）调用usb\_get\_device\_descriptor()函数读取设备描述字，写入usb\_dev.**descriptor**成员，函数内最终调用hc\_driver实例中的**hub\_control()**函数，实现控制传输（后面再介绍传输的实现）。

（4）调用usb\_new\_device (usb\_dev)函数注册表示根HUB的usb\_device实例。

下面将单独介绍注册usb\_device实例的usb\_new\_device (usb\_dev)函数实例，因为表示其它USB设备的usb\_device实例也需要调用此函数注册。

#### 2注册USB设备

usb\_new\_device ()函数用于注册表示USB设备的usb\_device实例，定义如下（/drivers/usb/core/hub.c）：

int usb\_new\_device(struct usb\_device \*udev)

{

int err;

if (udev->parent) {

device\_init\_wakeup(&udev->dev, 0);

}

/\* Tell the runtime-PM framework the device is active \*/

pm\_runtime\_set\_active(&udev->dev);

pm\_runtime\_get\_noresume(&udev->dev);

pm\_runtime\_use\_autosuspend(&udev->dev);

pm\_runtime\_enable(&udev->dev);

usb\_disable\_autosuspend(udev); /\*默认禁止所有设备自动睡眠\*/

err = **usb\_enumerate\_device**(udev); /\*主要是读取设备配置，/drivers/usb/core/hub.c\*/

...

**udev->dev.devt** = MKDEV(USB\_DEVICE\_MAJOR,

(((udev->bus->busnum-1) \* 128) + (udev->devnum-1))); /\*生成设备号\*/

announce\_device(udev); /\*输出信息，/drivers/usb/core/hub.c\*/

if (udev->serial) /\*输出设备信息\*/

add\_device\_randomness(udev->serial, strlen(udev->serial));

if (udev->product)

add\_device\_randomness(udev->product, strlen(udev->product));

if (udev->manufacturer)

add\_device\_randomness(udev->manufacturer,strlen(udev->manufacturer));

device\_enable\_async\_suspend(&udev->dev); /\*/include/linux/device.h\*/

/\* check whether the hub or firmware marks this port as non-removable \*/

if (udev->parent)

set\_usb\_port\_removable(udev); /\*设置设备接口移除\*/

err = **device\_add(&udev->dev)**; /\*添加设备，将创建设备文件\*/

...

/\* Create link files between child device and usb port device. 创建符号链接文件\*/

if (udev->parent) { /\*存在父设备\*/

struct usb\_hub \*hub = usb\_hub\_to\_struct\_hub(udev->parent); /\*上层HUB的usb\_hub实例\*/

int port1 = udev->portnum;

struct usb\_port \*port\_dev = hub->ports[port1 - 1];

err = sysfs\_create\_link(&udev->dev.kobj,&port\_dev->dev.kobj, "port");

...

err = sysfs\_create\_link(&port\_dev->dev.kobj,&udev->dev.kobj, "device");

...

if (!test\_and\_set\_bit(port1, hub->child\_usage\_bits))

pm\_runtime\_get\_sync(&port\_dev->dev);

}

(void) **usb\_create\_ep\_devs**(&udev->dev, &udev->ep0, udev);

/\*创建并注册表示端点的ep\_device实例，/drivers/usb/core/endpoint.c\*/

usb\_mark\_last\_busy(udev);

pm\_runtime\_put\_sync\_autosuspend(&udev->dev);

return err;

...

}

在调用usb\_new\_device ()函数注册usb\_device实例前，已经读取了设备描述字，函数内执行的主要工作如下：

（1）调用usb\_enumerate\_device()函数读取设备配置描述字。

（2）添加usb\_device实例内嵌device实例，添加时将会与通用USB设备驱动usb\_device\_driver实例usb\_generic\_driver匹配，并调用其probe()函数，probe()函数将会选择配置，读取接口描述字，创建并注册接口usb\_interface实例。

（3）调用usb\_create\_ep\_devs()函数创建表示端点0的ep\_device实例。

下面分别对以上三个步骤做简要介绍。

##### ■获取设备配置

usb\_enumerate\_device()函数主要用于获取设备配置，定义如下（/drivers/usb/core/hub.c）：

static int usb\_enumerate\_device(struct usb\_device \*udev)

{

int err;

struct usb\_hcd \*hcd = bus\_to\_hcd(udev->bus);

if (udev->config == NULL) {

err = **usb\_get\_configuration(udev)**;

/\*获取设备配置，存入usb\_device实例，/drivers/usb/core/config.c\*/

...

}

/\* read the standard strings and cache them if present \*/

udev->product = usb\_cache\_string(udev, udev->descriptor.iProduct);

udev->manufacturer = usb\_cache\_string(udev,udev->descriptor.iManufacturer);

udev->serial = usb\_cache\_string(udev, udev->descriptor.iSerialNumber);

...

return 0;

}

##### ■通用USB设备驱动

usb\_new\_device ()函数读取设备配置后，将添加udev->dev（device）实例，它将与usb\_generic\_driver通用USB设备驱动匹配，并调用其probe()函数，实例定义如下（/drivers/usb/core/generic.c）：

struct usb\_device\_driver usb\_generic\_driver = {

.name = "usb",

.probe = **generic\_probe**, /\*探测函数\*/

.disconnect = generic\_disconnect,

#ifdef CONFIG\_PM

.suspend = generic\_suspend,

.resume = generic\_resume,

#endif

.supports\_autosuspend = 1,

};

probe()探测函数generic\_probe()将选择设备配置，读取接口描述字，创建并注册接口usb\_interface实例，注册usb\_interface实例时将查找匹配的usb\_driver驱动，并调用其probe()函数，完成驱动程序的加载。

generic\_probe()函数代码简列如下（/drivers/usb/core/generic.c）：

static int generic\_probe(struct usb\_device \*udev)

{

int err, c;

if (udev->authorized == 0)

dev\_err(&udev->dev, "Device is not authorized for usage\n");

else {

c = **usb\_choose\_configuration(udev)**; /\*选择配置，返回配置编号，/drivers/usb/core/generic.c\*/

if (c >= 0) {

err = **usb\_set\_configuration(udev, c);**  /\*/drivers/usb/core/message.c\*/

/\*读取配置下的接口描述字，创建并注册接口usb\_interface实例\*/

...

}

}

/\* USB device state == configured ... usable \*/

usb\_notify\_add\_device(udev);

return 0;

}

usb\_choose\_configuration()函数用于选择设备配置，usb\_set\_configuration()函数读取配置下的接口描述字，创建并注册接口usb\_interface实例，其设备类型为usb\_if\_device\_type，源代码请读者自行阅读。

##### ■创建端点实例

usb\_new\_device()函数最后调用usb\_create\_ep\_devs()函数创建表示端点0的ep\_device实例，函数定义如下（/drivers/usb/core/endpoint.c）：

int usb\_create\_ep\_devs(struct device \*parent,struct usb\_host\_endpoint \*endpoint,struct usb\_device \*udev)

{

struct ep\_device \*ep\_dev; /\*/drivers/usb/core/endpoint.c\*/

int retval;

ep\_dev = kzalloc(sizeof(\*ep\_dev), GFP\_KERNEL);

...

ep\_dev->desc = &endpoint->desc;

ep\_dev->udev = udev;

ep\_dev->dev.groups = ep\_dev\_groups;

ep\_dev->dev.type = &usb\_ep\_device\_type;

ep\_dev->dev.parent = parent; /\*父设备为usb\_device实例\*/

dev\_set\_name(&ep\_dev->dev, "ep\_%02x", endpoint->desc.bEndpointAddress);

retval = **device\_register(&ep\_dev->dev);**

...

device\_enable\_async\_suspend(&ep\_dev->dev);

endpoint->ep\_dev = ep\_dev;

return retval;

...

}

创建端点ep\_device实例的主要用途是将端点信息（属性）导出到用户空间（sysfs文件系统）。

#### 3 HUB接口驱动

注册HUB设备usb\_device实例时，会匹配通用的usb设备驱动usb\_generic\_driver，在其prob()函数中将选择设备配置，读取接口描述字，创建并注册接口usb\_interface实例。在注册usb\_interface实例时会查找匹配的接口驱动usb\_driver实例，若匹配则调用usb\_driver实例中的probe()函数。

HUB接口匹配的usb\_driver实例为hub\_driver，这里的接口并不是HUB上的物理USB插口，而是其作为USB设备的逻辑接口（功能）。

hub\_driver实例定义如下（/drivers/usb/core/hub.c）：

static struct usb\_driver **hub\_driver** = {

.name = "hub",

.probe = **hub\_probe**, /\*探测函数\*/

.disconnect = hub\_disconnect,

.suspend = hub\_suspend,

.resume = hub\_resume,

.reset\_resume = hub\_reset\_resume,

.pre\_reset = hub\_pre\_reset,

.post\_reset = hub\_post\_reset,

.unlocked\_ioctl = hub\_ioctl,

.id\_table = **hub\_id\_table**, /\*匹配列表\*/

.supports\_autosuspend = 1,

};

hub\_driver驱动中匹配列表hub\_id\_table定义如下：

static const struct usb\_device\_id hub\_id\_table[] = {

{ .match\_flags = USB\_DEVICE\_ID\_MATCH\_VENDOR

| USB\_DEVICE\_ID\_MATCH\_INT\_CLASS,

.idVendor = USB\_VENDOR\_GENESYS\_LOGIC,

.bInterfaceClass = **USB\_CLASS\_HUB**, /\*HUB类\*/

.driver\_info = HUB\_QUIRK\_CHECK\_PORT\_AUTOSUSPEND},

{ .match\_flags = USB\_DEVICE\_ID\_MATCH\_DEV\_CLASS,

.bDeviceClass = **USB\_CLASS\_HUB**}, /\*HUB类\*/

{ .match\_flags = USB\_DEVICE\_ID\_MATCH\_INT\_CLASS,

.bInterfaceClass = **USB\_CLASS\_HUB**}, /\*HUB类\*/

{ }

};

在初始化函数usb\_hub\_init()中将注册hub\_driver实例，并创建工作队列，代码简列如下：

int usb\_hub\_init(void) /\*/drivers/usb/core/hub.c\*/

{

if (usb\_register(&hub\_driver) < 0) { /\*注册hub\_driver实例\*/

...

}

**hub\_wq** = alloc\_workqueue("usb\_hub\_wq", WQ\_FREEZABLE, 0); /\*创建工作队列\*/

if (hub\_wq)

return 0;

...

}

##### ■usb\_hub

在hub\_driver实例的prob()函数hub\_probe()中将创建并设置usb\_hub结构体实例，如下图所示：



usb\_hub结构体定义如下（/drivers/usb/core/hub.h）：

struct usb\_hub {

struct device \***intfdev**; /\*指向usb\_interface中device实例\*/

struct usb\_device \***hdev**; /\*指向表示HUB的usb\_device实例\*/

struct kref kref; /\*引用计数\*/

struct urb \***urb**; /\*用于中断轮询的urb\*/

/\* buffer for urb ... with extra space in case of babble \*/

u8 (\***buffer**)[8]; /\*传输数据缓存，8个缓存指针\*/

union {

struct usb\_hub\_status hub;

struct usb\_port\_status port;

}\*status; /\* buffer for status reports \*/

struct mutex status\_mutex; /\* for the status buffer \*/

int error; /\* last reported error \*/

int nerrors; /\* track consecutive errors \*/

/\*端口位标记\*/

unsigned long event\_bits[1]; /\* status change bitmask \*/

unsigned long change\_bits[1]; /\* ports with logical connectstatus change \*/

unsigned long removed\_bits[1]; /\* ports with a "removed"device present \*/

unsigned long wakeup\_bits[1]; /\* ports that have signaledremote wakeup \*/

unsigned long power\_bits[1]; /\* ports that are powered \*/

unsigned long child\_usage\_bits[1]; /\* ports powered on forchildren \*/

unsigned long warm\_reset\_bits[1]; /\* ports requesting warmreset recovery \*/

...

struct usb\_hub\_descriptor \***descriptor**; /\*指向HUB类描述符\*/

struct usb\_tt tt; /\*事务转换器\*/

unsigned mA\_per\_port; /\* current for each child \*/

#ifdef CONFIG\_PM

unsigned wakeup\_enabled\_descendants;

#endif

unsigned limited\_power:1;

unsigned quiescing:1;

unsigned disconnected:1;

unsigned in\_reset:1;

unsigned quirk\_check\_port\_auto\_suspend:1;

unsigned has\_indicators:1;

u8 indicator[USB\_MAXCHILDREN];

struct delayed\_work leds;

struct delayed\_work **init\_work**;

struct work\_struct  **events**; /\*探测接口连接，创建设备\*/

struct usb\_port \*\***ports**; /\*usb\_port结构体指针数组\*/

};

usb\_hub结构体中ports成员指向指针数组，数组项指向usb\_port实例，表示HUB中端口，usb\_port结构体定义如下（/drivers/usb/core/hub.h）：

struct usb\_port {

struct usb\_device \*child;

**struct device dev**;

struct usb\_dev\_state \*port\_owner;

struct usb\_port \*peer;

struct dev\_pm\_qos\_request \*req;

enum usb\_port\_connect\_type connect\_type;

usb\_port\_location\_t location;

struct mutex status\_lock;

u8 portnum; /\*端口编号\*/

unsigned int is\_superspeed:1;

};

##### ■探测函数

HUB接口驱动hub\_driver实例探测函数hub\_probe()代码简列如下（/drivers/usb/core/hub.c）：

static int hub\_probe(struct usb\_interface \*intf, const struct usb\_device\_id \*id)

{

struct usb\_host\_interface \*desc; /\*接口设置（接口描述字）\*/

struct usb\_endpoint\_descriptor \*endpoint; /\*端点描述字\*/

struct usb\_device \*hdev;

struct usb\_hub **\*hub**;

desc = intf->cur\_altsetting; /\*当前接口设置\*/

hdev = interface\_to\_usbdev(intf); /\*表示HUB的usb\_device实例\*/

#ifdef CONFIG\_PM

if (hdev->dev.power.autosuspend\_delay >= 0)

pm\_runtime\_set\_autosuspend\_delay(&hdev->dev, 0);

#endif

if (hdev->parent) { /\*非根HUB\*/

usb\_enable\_autosuspend(hdev);

} else { /\*根HUB\*/

const struct **hc\_driver** \*drv = bus\_to\_hcd(hdev->bus)->driver;　　/\*hc\_driver实例\*/

if (drv->bus\_suspend && drv->bus\_resume)

usb\_enable\_autosuspend(hdev);

}

if (hdev->level == MAX\_TOPO\_LEVEL) {

... /\*错误处理\*/

}

#ifdef CONFIG\_USB\_OTG\_BLACKLIST\_HUB

...

#endif

... /\*检测描述字错误\*/

/\*只能有一个端点？？\*/

if (desc->desc.bNumEndpoints != 1)

goto descriptor\_error;

endpoint = &desc->endpoint[0].desc; /\*端点0描述字\*/

if (!usb\_endpoint\_is\_int\_in(endpoint)) /\*端点0必须是中断输入，/include/uapi/linux/usb/ch9.h\*/

goto descriptor\_error;

dev\_info (&intf->dev, "USB hub found\n");

**hub = kzalloc(sizeof(\*hub), GFP\_KERNEL)**; /\*分配usb\_hub实例\*/

...

/\*初始化usb\_hub实例\*/

kref\_init(&hub->kref);

hub->intfdev = &intf->dev;

hub->hdev = **hdev**; /\*指向表示HUB设备的usb\_device实例\*/

INIT\_DELAYED\_WORK(&hub->leds, **led\_work**); /\*延时工作，/drivers/usb/core/hub.c\*/

INIT\_DELAYED\_WORK(&hub->init\_work, NULL);

INIT\_WORK(&hub->events, **hub\_event**); /\*工作执行函数为**hub\_event()**\*/

usb\_get\_intf(intf);

usb\_get\_dev(hdev);

usb\_set\_intfdata (intf, hub); /\*接口的驱动数据指向usb\_hub实例\*/

intf->needs\_remote\_wakeup = 1;

pm\_suspend\_ignore\_children(&intf->dev, true);

if (hdev->speed == USB\_SPEED\_HIGH)

highspeed\_hubs++;

if (id->driver\_info & HUB\_QUIRK\_CHECK\_PORT\_AUTOSUSPEND)

hub->quirk\_check\_port\_auto\_suspend = 1;

if (**hub\_configure**(hub, endpoint) >= 0) /\*HUB配置，/drivers/usb/core/hub.c\*/

return 0;

...

}

探测函数的主要工作是创建并设置usb\_hub实例，初始化events工作执行函数为**hub\_event()**（用于处理端口事件，如插入设备），获取hub端点0描述字，调用hub\_configure()函数配置HUB。

hub\_configure()函数主要工作是读取HUB描述字（类描述字），探测HUB端口，创建usb\_port实例，创建用于中断传输的urb实例等。下面将简要介绍hub\_configure()函数的实现。

###### **●配置HUB**

hub\_configure()函数代码简列如下（/drivers/usb/core/hub.c）：

static int hub\_configure(struct usb\_hub \*hub,struct usb\_endpoint\_descriptor \*endpoint)

/\*hub：指向usb\_hub实例，endpoint：指向端点0描述字\*/

{

struct usb\_hcd \*hcd;

struct usb\_device \*hdev = hub->hdev;

struct device \*hub\_dev = hub->intfdev;

...

hub->buffer = kmalloc(sizeof(\*hub->buffer), GFP\_KERNEL); /\*分配缓存指针数组\*/

...

hub->status = kmalloc(sizeof(\*hub->status), GFP\_KERNEL); /\*HUB状态\*/

...

hub->descriptor = kmalloc(sizeof(\*hub->descriptor), GFP\_KERNEL); /\*为HUB描述字分配空间\*/

....

ret = **get\_hub\_descripto**r(hdev, hub->descriptor); /\*读取HUB描述字\*/

...

maxchild = hub->descriptor->bNbrPorts; /\*端口数量\*/

...

hub->ports = kzalloc(maxchild \* sizeof(struct usb\_port \*), GFP\_KERNEL);

/\*分配端口usb\_port 实例指针数组\*/

...

spin\_lock\_init (&hub->tt.lock);

INIT\_LIST\_HEAD (&hub->tt.clear\_list);

INIT\_WORK(&hub->tt.clear\_work, hub\_tt\_work);

...

ret = usb\_get\_status(hdev, USB\_RECIP\_DEVICE, 0, &hubstatus);

...

hcd = bus\_to\_hcd(hdev->bus); /\*usb\_hcd\*/

... /\*设置电流\*/

ret = hub\_hub\_status(hub, &hubstatus, &hubchange); /\*HUB状态\*/

...

pipe = usb\_rcvintpipe(hdev, endpoint->bEndpointAddress);

maxp = usb\_maxpacket(hdev, pipe, usb\_pipeout(pipe));

if (maxp > sizeof(\*hub->buffer))

maxp = sizeof(\*hub->buffer);

**hub->urb = usb\_alloc\_urb**(0, GFP\_KERNEL); /\*分配urb实例，用于中断传输，见下文\*/

...

**usb\_fill\_int\_urb**(hub->urb, hdev, pipe, \*hub->buffer, maxp, **hub\_irq**,hub, endpoint->bInterval);

/\*填充urb实例\*/

if (hub->has\_indicators && blinkenlights)

hub->indicator[0] = INDICATOR\_CYCLE;

mutex\_lock(&usb\_port\_peer\_mutex);

for (i = 0; i < maxchild; i++) {

ret = **usb\_hub\_create\_port\_device**(hub, i + 1); /\*分配/设置/注册端口usb\_port 实例\*/

/\*/drivers/usb/core/port.c\*/

...

}

hdev->maxchild = i;

for (i = 0; i < hdev->maxchild; i++) {

struct usb\_port \*port\_dev = hub->ports[i];

pm\_runtime\_put(&port\_dev->dev);

}

...

if (hcd->driver->update\_hub\_device) {

ret = hcd->driver->update\_hub\_device(hcd, hdev,&hub->tt, GFP\_KERNEL);

...

}

usb\_hub\_adjust\_deviceremovable(hdev, hub->descriptor);

**hub\_activate(hub, HUB\_INIT);**  /\*激活HUB，/drivers/usb/core/hub.c\*/

return 0;

...

}

hub\_configure()函数读取HUB描述字，依此设置usb\_hub实例，创建并设置用于端点0中断传输的urb实例，最后调用hub\_activate()函数激活HUB及其端口。hub\_activate()函数用于分步骤激活HUB，函数源代码请读者自行阅读。

### 8.9.6数据传输

由前面的介绍我们知道，读取设备配置、描述字等信息时，就已经需要数据传输了，这里使用的是控制传输，数据传输由主机控制器实现。

在USB驱动程序中，传输的数据由**urb**结构体封装。USB驱动核心层提供了实现urb实例填充和提交的接口函数。

USB支持四种基本的数据传输模式：控制传输、同步传输、中断传输、批量传输。

每个数据传输都由urb结构体表示，发起传输的主要流程是：

（1）创建urb实例。

（2）填充urb实例。

（3）提交urb实例。

执行完以上3个步骤后，USB主机控制器会完成传输，完成后调用urb实例中的complete()回调函数。通常按照上面3个步骤执行的数据传输是异步传输，即调用者提交urb实例后，提交函数即返回，在传输完成后由USB驱动核心层调用urb实例中的complete()回调函数。

同步传输是指在提交urb实例后等待，等待传输完成后提交操作才返回。

下面先介绍urb结构体定义，然后介绍创建、填充、提交urb实例函数的实现，以及同步传输的接口函数。

#### 1 urb

USB总线每个数据传输由urb结构体表示（USB Request Block），结构体定义在/include/linux/usb.h头文件：

struct urb {

/\*USB驱动核心层和主机控制器私有数据\*/

struct kref kref; /\*引用计数\*/

void \*hcpriv; /\*主机控制器私有数据\*/

atomic\_t use\_count; /\* concurrent submissions counter \*/

atomic\_t reject; /\* submissions will fail \*/

int unlinked; /\* unlink error code \*/

/\*公共数据，驱动可使用的数据\*/

struct list\_head **urb\_list**; /\*双链表成员，将urb实例添加到管理双链表\*/

struct list\_head anchor\_list; /\* the URB may be anchored \*/

struct usb\_anchor \*anchor;

struct usb\_device **\*dev**; /\*所属USB设备的usb\_device实例\*/

struct usb\_host\_endpoint **\*ep;** /\*传输端点\*/

unsigned int **pipe**; /\*传输类型（管道）\*/

unsigned int stream\_id; /\*输入，流ID\*/

int status; /\* (return) non-ISO status \*/

unsigned int transfer\_flags; /\* (in) URB\_SHORT\_NOT\_OK | ...\*/

void \***transfer\_buffer**; /\*指向数据缓存区\*/

dma\_addr\_t transfer\_dma; /\* (in) dma addr for transfer\_buffer \*/

struct scatterlist \***sg**; /\* (in) scatter gather buffer list \*/

int num\_mapped\_sgs; /\* (internal) mapped sg entries \*/

int num\_sgs; /\*sg链表中实例数量\*/

u32 **transfer\_buffer\_length**; /\*数据缓存区长度\*/

u32 **actual\_length**; /\*返回值，表示实际传输的数据长度\*/

unsigned char \*setup\_packet; /\* (in) 控制传输设置阶段数据包\*/

dma\_addr\_t setup\_dma; /\* (in) dma addr for setup\_packet \*/

int start\_frame; /\* (modify) start frame (ISO) \*/

int number\_of\_packets; /\* (in) number of ISO packets \*/

int interval; /\* (modify) transfer interval\* (INT/ISO) ，传输间隔，中断或等时传输\*/

int error\_count; /\* (return) number of ISO errors \*/

void \*context; /\* (in) context for completion \*/

usb\_complete\_t **complete**; /\*urb传输完成后的回调函数\*/

struct usb\_iso\_packet\_descriptor iso\_frame\_desc[0];

};

urb结构体主要成员简介如下：

●**urb\_list**：双链表成员，将urb实例添加到管理双链表。

●**ep：**指向表示传输端点的usb\_host\_endpoint实例。

●**pipe：**表示传输端点、传输类型等参数，布局如下：



bit7：表示输出还是输入，0为输出，1为输入（主机视角）。

bit[8,14]：表示设备地址，bit[15,18]：端点编号。

bit[30,31]：表示传输类型，00为等时传输，01为中断传输，10为控制传输，11为批量传输。

在/include/linux/usb.h头文件定义了生成pipe值的函数，例如：

usb\_sndctrlpipe(dev, endpoint)：生成向设备dev，端点endpoint发起控制传输的pipe值。

●**transfer\_buffer：**数据缓存区指针，缓存区必须是动态分配的。

●**transfer\_flags**：传输标记，取值定义如下（/include/linux/usb.h）：

#define URB\_SHORT\_NOT\_OK 0x0001 /\* report short reads as errors \*/

#define URB\_ISO\_ASAP 0x0002 /\*适用于等时传输\*/

#define URB\_NO\_TRANSFER\_DMA\_MAP 0x0004 /\* urb->transfer\_dma valid on submit \*/

#define URB\_NO\_FSBR 0x0020 /\* UHCI-specific \*/

#define URB\_ZERO\_PACKET 0x0040 /\* Finish bulk OUT with short packet \*/

#define URB\_NO\_INTERRUPT 0x0080 /\*\*/

#define URB\_FREE\_BUFFER 0x0100 /\* Free transfer buffer with the URB \*/

...

●**complete：**urb传输完成的回调函数指针，函数原型为（/include/linux/usb.h）：

typedef void (\*usb\_complete\_t)(struct urb \*);

#### 2分配urb实例

内核在/drivers/usb/core/urb.c文件内定义了创建urb实例的接口函数**usb\_alloc\_urb**()，定义如下：

struct urb \*usb\_alloc\_urb(int iso\_packets, gfp\_t mem\_flags)

/\*iso\_packets：等时数据包数量，中断、控制、批量传输时为0，mem\_flags：内存分配标记\*/

{

struct urb \*urb;

urb = kmalloc(sizeof(struct urb) +

iso\_packets \* sizeof(struct usb\_iso\_packet\_descriptor),mem\_flags); /\*分配urb实例\*/

...

**usb\_init\_urb(urb)**; /\*初始化urb实例，清零、初始化双链表成员等\*/

return urb; /\*返回urb实例指针\*/

}

void **usb\_free\_urb**(struct urb \*urb)：释放urb实例。

#### 3填充urb实例

内核在/include/linux/usb.h头文件定义了填充urb实例的接口函数。

填充控制传输urb实例的函数如下所示：

static inline void usb\_fill\_control\_urb(struct urb \*urb,struct usb\_device \*dev,unsigned int pipe,

unsigned char \*setup\_packet,void \*transfer\_buffer,int buffer\_length,

usb\_complete\_t complete\_fn,void \*context)

/\*setup\_packet：指向设置阶段数据缓存，transfer\_buffer：数据阶段缓存， complete\_fn：回调函数\*/

{

urb->dev = dev;

urb->pipe = pipe;

urb->setup\_packet = setup\_packet;

urb->transfer\_buffer = transfer\_buffer;

urb->transfer\_buffer\_length = buffer\_length;

urb->complete = complete\_fn;

urb->context = context;

}

填充批量传输urb实例的函数如下：

static inline void usb\_fill\_bulk\_urb(struct urb \*urb,struct usb\_device \*dev,unsigned int pipe,

void \*transfer\_buffer,int buffer\_length,usb\_complete\_t complete\_fn,void \*context)

{

urb->dev = dev;

urb->pipe = pipe;

urb->transfer\_buffer = transfer\_buffer;

urb->transfer\_buffer\_length = buffer\_length;

urb->complete = complete\_fn;

urb->context = context;

}

填充中断传输urb实例的函数如下：

static inline void usb\_fill\_int\_urb(struct urb \*urb,struct usb\_device \*dev,unsigned int pipe,

void \*transfer\_buffer,int buffer\_length,usb\_complete\_t complete\_fn,

void \*context,int **interval**)

{

urb->dev = dev;

urb->pipe = pipe;

urb->transfer\_buffer = transfer\_buffer;

urb->transfer\_buffer\_length = buffer\_length;

urb->complete = complete\_fn;

urb->context = context;

if (dev->speed == USB\_SPEED\_HIGH || dev->speed == USB\_SPEED\_SUPER) {

interval = clamp(interval, 1, 16);

urb->interval = 1 << (interval - 1);

} else {

urb->interval = interval; /\*传输间隔\*/

}

urb->start\_frame = -1;

}

#### 4提交urb实例

usb\_submit\_urb()函数用于提交urb实例，函数代码简列如下（/drivers/usb/core/urb.c）：

int usb\_submit\_urb(struct urb \*urb, gfp\_t mem\_flags)

{

static int pipetypes[4] = { /\*传输（管道类型）\*/

PIPE\_CONTROL, PIPE\_ISOCHRONOUS, PIPE\_BULK, PIPE\_INTERRUPT

};

int xfertype, max;

struct usb\_device \*dev; /\*USB设备\*/

struct usb\_host\_endpoint \*ep; /\*端点\*/

int is\_out;

unsigned int allowed;

if (!urb || !**urb->complete**) /\*urb必须定义complete()函数\*/

return -EINVAL;

...

dev = urb->dev; /\*usb\_device实例\*/

if ((!dev) || (dev->state < USB\_STATE\_UNAUTHENTICATED))

return -ENODEV;

ep = **usb\_pipe\_endpoint**(dev, urb->pipe); /\*获取传输端点信息，/include/linux/usb.h\*/

... /\*错误处理\*/

urb->ep = ep;

urb->status = -EINPROGRESS;

urb->actual\_length = 0;

/\*安全检查\*/

xfertype = usb\_endpoint\_type(&ep->desc);

if (xfertype == USB\_ENDPOINT\_XFER\_CONTROL) { /\*控制端点\*/

struct usb\_ctrlrequest \*setup =(struct usb\_ctrlrequest \*) urb->setup\_packet;

if (!setup)

return -ENOEXEC;

is\_out = !(setup->bRequestType & USB\_DIR\_IN) ||!setup->wLength;

} else {

is\_out = usb\_endpoint\_dir\_out(&ep->desc);

}

/\* Clear the internal flags and cache the direction for later use \*/

urb->transfer\_flags &= ~(URB\_DIR\_MASK | URB\_DMA\_MAP\_SINGLE |

URB\_DMA\_MAP\_PAGE | URB\_DMA\_MAP\_SG | URB\_MAP\_LOCAL |

URB\_SETUP\_MAP\_SINGLE | URB\_SETUP\_MAP\_LOCAL |

URB\_DMA\_SG\_COMBINED);

urb->transfer\_flags |= (is\_out ? URB\_DIR\_OUT : URB\_DIR\_IN); /\*传输标记\*/

if (xfertype != USB\_ENDPOINT\_XFER\_CONTROL &&dev->state < USB\_STATE\_CONFIGURED)

return -ENODEV;

max = usb\_endpoint\_maxp(&ep->desc); /\*端点最大传输数据大小\*/

...

/\*周期传输（中断、等时）中一个帧或微帧内的数据限制/

if (xfertype == USB\_ENDPOINT\_XFER\_ISOC) {

int n, len;

/\* SuperSpeed isoc endpoints have up to 16 bursts of up to

\* 3 packets each

\*/

if (dev->speed == USB\_SPEED\_SUPER) {

int burst = 1 + ep->ss\_ep\_comp.bMaxBurst;

int mult = USB\_SS\_MULT(ep->ss\_ep\_comp.bmAttributes);

max \*= burst;

max \*= mult;

}

/\* "high bandwidth" mode, 1-3 packets/uframe? \*/

if (dev->speed == USB\_SPEED\_HIGH) {

int mult = 1 + ((max >> 11) & 0x03);

max &= 0x07ff;

max \*= mult;

}

if (urb->number\_of\_packets <= 0)

return -EINVAL;

for (n = 0; n < urb->number\_of\_packets; n++) {

len = urb->iso\_frame\_desc[n].length;

if (len < 0 || len > max)

return -EMSGSIZE;

urb->iso\_frame\_desc[n].status = -EXDEV;

urb->iso\_frame\_desc[n].actual\_length = 0;

}

} else if (urb->num\_sgs && !urb->dev->bus->no\_sg\_constraint &&

dev->speed != USB\_SPEED\_WIRELESS) {

struct scatterlist \*sg;

int i;

for\_each\_sg(urb->sg, sg, urb->num\_sgs - 1, i)

if (sg->length % max)

return -EINVAL;

}

/\* the I/O buffer must be mapped/unmapped, except when length=0 \*/

if (urb->transfer\_buffer\_length > INT\_MAX) /\*缓存区长度不能超过限制值，/include/linux/kernel.h\*/

return -EMSGSIZE;

/\*类型检查\*/

if (usb\_pipetype(urb->pipe) != pipetypes[xfertype])

dev\_WARN(&dev->dev, "BOGUS urb xfer, pipe %x != type %x\n",

usb\_pipetype(urb->pipe), pipetypes[xfertype]);

/\* Check against a simple/standard policy \*/

allowed = (URB\_NO\_TRANSFER\_DMA\_MAP | URB\_NO\_INTERRUPT | URB\_DIR\_MASK |

URB\_FREE\_BUFFER);

switch (xfertype) { /\*传输类型\*/

case USB\_ENDPOINT\_XFER\_BULK: /\*批量和中断\*/

case USB\_ENDPOINT\_XFER\_INT:

if (is\_out)

allowed |= URB\_ZERO\_PACKET;

/\* FALLTHROUGH \*/

case USB\_ENDPOINT\_XFER\_CONTROL: /\*控制传输\*/

allowed |= URB\_NO\_FSBR; /\* only affects UHCI \*/

/\* FALLTHROUGH \*/

default: /\* all non-iso endpoints \*/

if (!is\_out)

allowed |= URB\_SHORT\_NOT\_OK;

break;

case USB\_ENDPOINT\_XFER\_ISOC:

allowed |= URB\_ISO\_ASAP;

break;

}

allowed &= urb->transfer\_flags;

if (allowed != urb->transfer\_flags)

dev\_WARN(&dev->dev, "BOGUS urb flags, %x --> %x\n",urb->transfer\_flags, allowed);

/\*使周期传输的间隔合法\*/

switch (xfertype) {

case USB\_ENDPOINT\_XFER\_ISOC:

case USB\_ENDPOINT\_XFER\_INT:

/\* too small? \*/

switch (dev->speed) {

case USB\_SPEED\_WIRELESS:

if ((urb->interval < 6)&& (xfertype == USB\_ENDPOINT\_XFER\_INT))

return -EINVAL;

default:

if (urb->interval <= 0)

return -EINVAL;

break;

}

/\* too big? \*/

switch (dev->speed) {

case USB\_SPEED\_SUPER: /\* units are 125us \*/

/\* Handle up to 2^(16-1) microframes \*/

if (urb->interval > (1 << 15))

return -EINVAL;

max = 1 << 15;

break;

case USB\_SPEED\_WIRELESS:

if (urb->interval > 16)

return -EINVAL;

break;

case USB\_SPEED\_HIGH: /\* units are microframes \*/

/\* NOTE usb handles 2^15 \*/

if (urb->interval > (1024 \* 8))

urb->interval = 1024 \* 8;

max = 1024 \* 8;

break;

case USB\_SPEED\_FULL: /\* units are frames/msec \*/

case USB\_SPEED\_LOW:

if (xfertype == USB\_ENDPOINT\_XFER\_INT) {

if (urb->interval > 255)

return -EINVAL;

/\* NOTE ohci only handles up to 32 \*/

max = 128;

} else {

if (urb->interval > 1024)

urb->interval = 1024;

/\* NOTE usb and ohci handle up to 2^15 \*/

max = 1024;

}

break;

default:

return -EINVAL;

}

if (dev->speed != USB\_SPEED\_WIRELESS) {

/\* Round down to a power of 2, no more than max \*/

urb->interval = min(max, 1 << ilog2(urb->interval));

}

}

return **usb\_hcd\_submit\_urb**(urb, mem\_flags); /\*向HCD提交urb，/drivers/usb/core/hcd.c\*/

}

简而言之，usb\_submit\_urb()函数的主要工作就是对urb实例进行检查，合格后调用usb\_hcd\_submit\_urb()函数将urb实例提交给主机控制器。

##### ■urb提交给主机控制器

usb\_hcd\_submit\_urb()函数定义在/drivers/usb/core/hcd.c文件内，代码简列如下：

int usb\_hcd\_submit\_urb (struct urb \*urb, gfp\_t mem\_flags)

{

int status;

struct usb\_hcd \*hcd = bus\_to\_hcd(urb->dev->bus); /\*主机控制器\*/

/\*增加引用计数\*/

usb\_get\_urb(urb);

atomic\_inc(&urb->use\_count);

atomic\_inc(&urb->dev->urbnum);

usbmon\_urb\_submit(&hcd->self, urb); /\*可能为空，/include/linux/usb/hcd.h\*/

if (is\_root\_hub(urb->dev)) { /\*与根HUB传输数据\*/

status = **rh\_urb\_enqueue(hcd, urb)**; /\*实例控制传输等，/drivers/usb/core/hcd.c\*/

} else { /\*不是与根HUB传输数据\*/

status = map\_urb\_for\_dma(hcd, urb, mem\_flags);

if (likely(status == 0)) {

**status = hcd->driver->urb\_enqueue(hcd, urb, mem\_flags)**;

if (unlikely(status))

unmap\_urb\_for\_dma(hcd, urb);

}

}

if (unlikely(status)) {

usbmon\_urb\_submit\_error(&hcd->self, urb, status);

urb->hcpriv = NULL;

INIT\_LIST\_HEAD(&urb->urb\_list);

atomic\_dec(&urb->use\_count);

atomic\_dec(&urb->dev->urbnum);

if (atomic\_read(&urb->reject))

wake\_up(&usb\_kill\_urb\_queue);

usb\_put\_urb(urb);

}

return status;

}

usb\_hcd\_submit\_urb()函数对urb的提交要分情况而定，如下所示（如下图所示）：

1、与根HUB的数据传输

（1）中断传输：提交到端点0的urb双链表，激活usb\_hcd轮询定时器rh\_timer，以探测根HUB状态变化。

（2）控制传输：提交到端点0的urb双链表，并立即调用hcd->driver->hub\_control()函数执行控制传输，并处理传输结果。

2、与非根HUB的数据传输

调用hcd->driver->urb\_enqueue()函数将urb添加到端点的urb双链表。



urb添加到端点urb实例双链表后，由主机控制器驱动程序负责调度，完成数据的传输。

完成传输的urb实例将从端点urb双链表中移出，添加到usb\_hcd实例中的giveback\_urb\_bh结构体中的双链表，随后将通过tasklet机制，调用urb实例中的complete()函数，执行传输完成后的工作。

在添加usb\_hcd实例的usb\_add\_hcd()函数中，将初始化其high\_prio\_bh和low\_prio\_bh成员，两个成员中内嵌tasklet实例的执行函数为usb\_giveback\_urb\_bh()，此函数扫描giveback\_urb\_bh实例中urb双链表，释放urb实例，调用其complete()函数等。

#### 5同步传输

下面几个函数用于发送同步型数据，函数内将创建urb实例，并将其发送出去，等待传输完成，函数返回。这些函数可能用在中断上下文中（/drivers/usb/core/message.c）。

● int **usb\_control\_msg**(struct usb\_device \*dev, unsigned int pipe, \

\_\_u8 request, \_\_u8 requesttype, \_\_u16 value, \_\_u16 index,void \*data, \_\_u16 size, int timeout);

●int **usb\_interrupt\_msg**(struct usb\_device \*usb\_dev, unsigned int pipe,void \*data, int len, \

int \*actual\_length, int timeout);

● int **usb\_bulk\_msg**(struct usb\_device \*usb\_dev, unsigned int pipe, \

void \*data, int len, int \*actual\_length,int timeout);

### 8.9.7探测新设备

至此，似乎USB总线驱动和数据传输都介绍完了，但是还有一个重要的问题还没弄明白。那就是当向USB端口插入新设备时，主机控制器是如何检测到的？

外部设备插入HUB端口时，HUB会检测到电压变化，感知到有设备插入，然后通过端点0的中断传输将端口的变化传输给主机控制器。这个传输不是HUB主动发起的，而是在主机控制器轮询时发起的。

下面用图示的方法简要说明探测新设备的过程，如下图所示：



在创建根HUB的usb\_hub实例时，创建了用于中断传输的urb实例，其complete()函数为**hub\_irq()**，工作成员events的执行函数设为hub\_event()，这是探测设备的函数。

在创建主机控制器usb\_hcd时，其轮询定时器rh\_timer执行函数设为**rh\_timer\_func**()，在添加usb\_hcd实例时，会激活此定时器。

轮询定时器到期执行函数rh\_timer\_func()将检测根HUB的状态变化，如果有状态变化则将根HUB的中断传输urb实例关联到usb\_hcd->status\_urb成员，并添加到usb\_hcd实例的giveback\_urb\_bh结构体成员的双链表中，即将其视为传输完成的urb，随后将调用urb的complete()函数**hub\_irq()**。

hub\_irq()函数将调用kick\_hub\_wq()函数，kick\_hub\_wq()函数将激活usb\_hub实例中events工作，其执行函数为hub\_event()。

hub\_event()函数检测HUB端口的变化，为端口连接的设备创建并注册usb\_device实例，函数调用关系简列如下图所示：



普通设备usb\_device实例的创建和注册与前面介绍的根HUB设备的处理相似，usb\_device实例将匹配通用的USB设备驱动usb\_generic\_driver，在其probe()函数中读取设备配置和接口信息，创建并注册表示接口的usb\_interface实例，usb\_interface实例将匹配用户注册的接口驱动程序。用户实际要编写的是接口驱动程序。

由于作者水平有限，只是简单介绍了一下探测设备的流程，具体函数源代码请读者自行阅读。

### 8.9.8接口驱动示例

内核在/drivers/usb/usb-skeleton.c文件内实现了一个USB接口驱动的示例程序，用户可依此模板编写USB接口驱动程序。下面对此示例程序做简要介绍。

#### 1驱动框架

在/drivers/usb/usb-skeleton.c文件内实现的USB接口驱动框架如下图所示。

内核提供了一个将连接到USB总线上的设备视为字符设备的驱动程序框架，即所有USB设备共用一个主设备号USB\_MAJOR（180），不同的设备分配不同的从设备号。

在/drivers/usb/core/file.c文件内定义的初始化函数usb\_major\_init()（由usb\_init()函数调用），将创建并注册了对应的cdev实例，关联的file\_operations结构体实例为**usb\_fops**。usb\_fops实例中只定义了open()函数，函数内依据从设备号查找file\_operations结构体指针数组usb\_minors[]，将其指向的file\_operations实例赋予file实例。接口驱动程序在定义file\_operations实例后，需要依从设备号关联到对应的usb\_minors[]数组项，从设备号是动态分配的。



接口驱动程序需定义usb\_class\_driver结构体实例，以此来注册定义的file\_operations实例，结构体定义如下（/include/linux/usb.h）：

struct usb\_class\_driver {

char \*name;

char \*(\*devnode)(struct device \*dev, umode\_t \*mode); /\*设备文件名称\*/

const struct file\_operations \***fops**; /\*文件操作结构\*/

int minor\_base; /\*动态分配从设备号的起点\*/

};

接口函数int **usb\_register\_dev**(struct usb\_interface \*intf,struct usb\_class\_driver \*class\_driver)定义在文件/drivers/usb/core/file.c，函数内为USB设备创建设备类class实例，为接口分配从设备号，将class\_driver->fops指向实例关联到usb\_minors[]数组中从设备号对应项，为接口创建并添加device实例，关联到其usb\_dev成员，源代码请读者自行阅读。

在添加device实例时，将为接口创建设备文件，文件名称来源的顺序依次为：

（1）usb\_class\_driver->devnode()。

（2）dev->init\_name 或dev->kobject名称。

在usb\_register\_dev()函数中将设备dev->kobject名称为“class\_driver->name”后接从设备号相对于minor\_base的偏移量，如果没有定义usb\_class\_driver->devnode()函数，这将做为接口的设备文件名称。

在usb-skeleton.c文件内接口驱动的probe()函数中，将为接口创建了usb\_skel结构体实例，用来搜集接口的信息，其中包含一个用于批量传输的urb实例；调用usb\_register\_dev()接口函数创建并添加device实例，注册定义的file\_operations实例skel\_fops等。

usb\_class\_driver结构体实例skel\_class定义如下：

static struct usb\_class\_driver **skel\_class** = {

.name = "**skel%d**", /\*设备文件名称\*/

.fops = &**skel\_fops,**

.minor\_base = USB\_SKEL\_MINOR\_BASE,

};

文件操作结构skel\_fops实例中的函数通过此批量传输urb实现数据的传输。

#### 2探测函数

接口驱动定义如下：

static struct usb\_driver skel\_driver = {

.name = "skeleton",

.probe = **skel\_probe**,

.disconnect = skel\_disconnect,

.suspend = skel\_suspend,

.resume = skel\_resume,

.pre\_reset = skel\_pre\_reset,

.post\_reset = skel\_post\_reset,

.id\_table = **skel\_table**, /\*匹配厂商和产品ID\*/

.supports\_autosuspend = 1,

};

module\_usb\_driver(skel\_driver);

probe()探测函数skel\_probe()代码简列如下：

static int skel\_probe(struct usb\_interface \*interface,const struct usb\_device\_id \*id)

{

struct usb\_skel \*dev;

struct usb\_host\_interface \*iface\_desc;

struct usb\_endpoint\_descriptor \*endpoint;

size\_t buffer\_size;

int i;

int retval = -ENOMEM;

dev = kzalloc(sizeof(\*dev), GFP\_KERNEL); /\*分配usb\_skel实例\*/

...

kref\_init(&dev->kref);

sema\_init(&dev->limit\_sem, WRITES\_IN\_FLIGHT);

mutex\_init(&dev->io\_mutex);

spin\_lock\_init(&dev->err\_lock);

init\_usb\_anchor(&dev->submitted);

init\_waitqueue\_head(&dev->bulk\_in\_wait);

dev->udev = usb\_get\_dev(interface\_to\_usbdev(interface)); /\*usb\_device实例\*/

dev->interface = interface; /\*接口\*/

iface\_desc = interface->cur\_altsetting;

for (i = 0; i < iface\_desc->desc.bNumEndpoints; ++i) { /\*查找第一个批理输入、输出端点\*/

endpoint = &iface\_desc->endpoint[i].desc; /\*端点描述字\*/

if (!dev->bulk\_in\_endpointAddr &&usb\_endpoint\_is\_bulk\_in(endpoint)) {

/\* we found a bulk in endpoint \*/

buffer\_size = usb\_endpoint\_maxp(endpoint);

dev->bulk\_in\_size = buffer\_size;

dev->bulk\_in\_endpointAddr = endpoint->bEndpointAddress;

dev->**bulk\_in\_buffer = kmalloc(buffer\_size, GFP\_KERNEL)**; /\*分配缓存区\*/

...

dev->bulk\_in\_urb = usb\_alloc\_urb(0, GFP\_KERNEL); /\*分配批量传输urb\*/

...

}

if (!dev->bulk\_out\_endpointAddr &&usb\_endpoint\_is\_bulk\_out(endpoint)) {

/\* we found a bulk out endpoint \*/

dev->bulk\_out\_endpointAddr = endpoint->bEndpointAddress;

}

}

... /\*错误处理\*/

usb\_set\_intfdata(interface, dev); /\*interface->dev.driver\_data=usb\_skel\*/

retval = **usb\_register\_dev**(interface, &**skel\_class**); /\*前面介绍过的接口函数\*/

...

return 0;

...

}

#### 3文件操作函数

usb\_class\_driver结构体实例skel\_class中关联的file\_operations实例**skel\_fops**定义如下：

static const struct file\_operations skel\_fops = { /\*操作设备文件\*/

.owner = THIS\_MODULE,

.read = **skel\_read**,

.write = skel\_write,

.open = **skel\_open**,

.release = skel\_release,

.flush = skel\_flush,

.llseek = noop\_llseek,

};

skel\_fops实例中的函数是用户操作接口设备文件的接口，下面简单介绍一下其中的open()函数和read()函数，其它函数请读者自行阅读源代码。

##### ■打开函数

打开函数skel\_open()代码简列如下：

static int skel\_open(struct inode \*inode, struct file \*file)

{

struct usb\_skel \*dev;

struct usb\_interface \*interface;

int subminor;

int retval = 0;

subminor = iminor(inode); /\*从设备号\*/

interface = usb\_find\_interface(&skel\_driver, subminor); /\*接口\*/

...

dev = usb\_get\_intfdata(interface); /\*usb\_skel实例\*/

...

retval = usb\_autopm\_get\_interface(interface); /\*/drivers/usb/core/driver.c\*/

...

/\* increment our usage count for the device \*/

kref\_get(&dev->kref);

/\* save our object in the file's private structure \*/

**file->private\_data = dev**; /\*file私有数据指向usb\_skel实例\*/

exit:

return retval;

}

##### ■读函数

在看skel\_read()函数前先看一下usb\_skel结构体的定义：

struct usb\_skel {

struct usb\_device \*udev; /\*指向USB设备usb\_device实例\*/

struct usb\_interface \*interface; /\*接口\*/

struct semaphore limit\_sem; /\* limiting the number of writes in progress \*/

struct usb\_anchor submitted; /\* in case we need to retract our submissions \*/

struct urb \***bulk\_in\_urb**; /\*指向读操作urb，写操作另外创建urb\*/

unsigned char \***bulk\_in\_buffer**; /\*接收（读）数据缓存区（urb中缓存区）\*/

size\_t bulk\_in\_size; /\*缓存区长度\*/

size\_t bulk\_in\_filled; /\*缓存区中已填充长度\*/

size\_t bulk\_in\_copied; /\*已复制到用户空间数据长度\*/

\_\_u8 bulk\_in\_endpointAddr; /\*批量输入端点编号\*/

\_\_u8 bulk\_out\_endpointAddr; /\*批量输出端点编号\*/

int errors; /\* the last request tanked \*/

bool ongoing\_read; /\*正在读\*/

spinlock\_t err\_lock; /\* lock for errors \*/

struct kref kref;

struct mutex io\_mutex; /\* synchronize I/O with disconnect \*/

wait\_queue\_head\_t bulk\_in\_wait; /\*读等待队列\*/

};

读函数流程如下图所示，usb\_skel实例中bulk\_in\_buffer成员关联到读urb的缓存区，若缓存区有数据且长度够，则直接将缓存区数据复制到用户空间。若缓存区数据不够（或没有数据），则调用skel\_do\_read\_io()函数从设备读数据，操作函数睡眠等待。skel\_do\_read\_io()函数读到数据后，urb实例回调函数将唤醒读操作睡眠的进程，唤醒后从缓存区复制数据到用户空间。如果有其它进程在进行读操作，当前进程也将睡眠等待。



读函数skel\_read()代码简列如下：

static ssize\_t skel\_read(struct file \*file, char \*buffer, size\_t count,loff\_t \*ppos)

{

struct usb\_skel \*dev;

int rv;

bool ongoing\_io;

dev = file->private\_data; /\*usb\_skel实例\*/

/\*不能读\*/

if (!dev->bulk\_in\_urb || !count)

return 0;

/\* no concurrent readers \*/

rv = mutex\_lock\_interruptible(&dev->io\_mutex);

if (rv < 0)

return rv;

if (!dev->interface) { /\* disconnect() was called \*/

rv = -ENODEV;

goto exit;

}

/\* if IO is under way, we must not touch things \*/

retry:

spin\_lock\_irq(&dev->err\_lock);

ongoing\_io = dev->ongoing\_read;

spin\_unlock\_irq(&dev->err\_lock);

if (ongoing\_io) { /\*正在读，需要睡眠等待\*/

/\* nonblocking IO shall not wait \*/

if (file->f\_flags & O\_NONBLOCK) {

rv = -EAGAIN;

goto exit;

}

/\*

\* IO may take forever

\* hence wait in an interruptible state

\*/

rv = **wait\_event\_interruptible**(dev->bulk\_in\_wait, (!dev->ongoing\_read));

if (rv < 0)

goto exit;

}

/\* errors must be reported \*/

... /\*报告错误\*/

if (dev->bulk\_in\_filled) { /\*缓存区有数据，直接复制\*/

/\* we had read data \*/

size\_t available = dev->bulk\_in\_filled - dev->bulk\_in\_copied;

size\_t chunk = min(available, count);

if (!available) {

rv = **skel\_do\_read\_io**(dev, count); /\*读数据到缓存区\*/

if (rv < 0)

goto exit;

else

goto retry;

}

/\*缓存区有数据，复制到用户空间\*/

if (**copy\_to\_user**(buffer,dev->bulk\_in\_buffer + dev->bulk\_in\_copied,chunk))

rv = -EFAULT;

else

rv = chunk;

dev->bulk\_in\_copied += chunk;

/\*数据不够，再读\*/

if (available < count)

skel\_do\_read\_io(dev, count - chunk);

} else { /\*缓存区没有数据，读数据\*/

rv = skel\_do\_read\_io(dev, count);

if (rv < 0)

goto exit;

else

goto retry;

}

exit:

mutex\_unlock(&dev->io\_mutex);

return rv;

}

skel\_do\_read\_io()函数用于通过USB总线从设备读数据，函数定义如下：

static int skel\_do\_read\_io(struct usb\_skel \*dev, size\_t count)

{

int rv;

/\*填充urb\*/

**usb\_fill\_bulk\_urb**(dev->bulk\_in\_urb,

dev->udev,

usb\_rcvbulkpipe(dev->udev,dev->bulk\_in\_endpointAddr),

dev->bulk\_in\_buffer,

min(dev->bulk\_in\_size, count),

**skel\_read\_bulk\_callback**, /\*回调函数，唤醒睡眠等待读进程等\*/

dev);

/\* tell everybody to leave the URB alone \*/

spin\_lock\_irq(&dev->err\_lock);

dev->ongoing\_read = 1;

spin\_unlock\_irq(&dev->err\_lock);

/\* submit bulk in urb, which means no data to deliver \*/

dev->bulk\_in\_filled = 0;

dev->bulk\_in\_copied = 0;

/\*提交urb，执行数据传输\*/

rv = **usb\_submit\_urb**(dev->bulk\_in\_urb, GFP\_KERNEL);

...

return rv;

}

skel\_fops实例中其它操作函数请读者自行阅读源代码。

至此，USB总线及设备驱动都介绍完了，对于通用（标准）的USB设备，内核都已经实现了其驱动，用户只需要在配置内核时选择即可。移植内核时主要是要移植USB主机控制器驱动。

## 8.10模块

模块是内核编译源文件产生的一种目标文件，用于向内核动态添加驱动程序、文件系统类型等组件，内核在运行过程中加载模块后，无需重启即可使用模块中的代码，模块也可以被动态卸载。模块加载到内核地址空间的VMLLOC区域，模块的动态加/卸载特性使内核具有了扩展性，内核映象可保持一个较小的尺寸，在运行时根据需要加卸载相关的功能模块。

在编译内核的过程中可以配置选择驱动程序或文件系统等代码是永久编译入内核还是以模块的形式编译，编译成模块时，产生的是.ko形式的可重定位目标文件。用户进程通过系统调用加载模块时，内核将模块代码加载到内核VMALLOC区，并解决引用问题。所谓引用就是模块中引用内核或其它模块内的函数或全局变量，模块在编译时不知道它们的地址，在加载模块时内核需要将地址告知模块。

模块的加卸载由用户进程通过系统调用完成，内核自身触发模块加卸载时，将信息传递给用户态守护进程，由用户进程完成加卸载工作。

本节首先简要介绍内核镜像、模块等文件格式，然后介绍模块的编译，最后介绍内核对模块的管理加卸载操作。

### 8.10.1目标文件格式

ELF(Executable and Linkable Format)是一种可执行文件、目标文件和库文件使用的标准文件格式，在Linux下成为标准格式已经很长时间了。内核映像文件、可重定位文件、模块文件都使用的是ELF格式。

#### 1 ELF文件格式

内核源代码本质上是一个大的程序，各源代码文件经过选择，编译生成目标文件，所有的目标文件链接成单一的可执行文件。

模块文件只编译成目标文件，不参与链接，在运行时动态加载到内核地址空间，解决与内核代码之间的引用关系。

目标文件（包括模块）和内核可执行文件都采用ELF文件格式，ELF文件各个部分组成如下图所示（第5章有更详细的介绍）：



源文件编译生成的目标文件结构通常是链接视图，文件内容包括ELF文件头、节、节头表等。

ELF文件头包含文件类型和大小的有关信息、节头表位置和长度等信息。节（section）就是源代码中定义的.text节（代码）、.rodata节（只读数据）中的内容，即指令和数据。节头表中每一个表项描述了一个节的信息。

各目标文件通过链接，拼接在一起，生成可执行文件。各目标文件中相同类型的节在可执行文件中放在一起，如所有目标文件中的.text放在一起，形成段。段可以理解成是对相同属性节的归类和整理。

在可执行视图中，ELF文件头中包含了段表（程序头表）的信息，段表中每一个表项描述了一个段的信息。段的类型有INTERP、LOAD、DYNAMIC等，其中LOAD是表示必须映射到虚拟地址空间的段，其它段保存的是其它信息，不需要映射到虚拟地址空间。LOAD段中包含.text和.rodata等节，它们都是要映射到虚拟地址空间中的节。

内核源代码文件选择编译成模块时，只编译生成可重定位目标文件，不参与内核可执行文件的链接。

可重定位文件中没有使用绝对地址，使用的都是相对地址，因此目标文件可加载到任意地址运行。但是，模块会引用内核和其它模块定义的函数和变量，因此模块加载后需要解决引用问题。

#### 2内核导出符号

ELF文件内的符号表节，保存了本程序定义和引用的所有全局变量和函数地址。如果程序引用了一个自身代码没有定义的符号，则称之为未定义符号。

用户空间工具nm可生成目标文件中定义和使用的所有符号列表，如：

00000000 T add

**U exit**

0000001a T main

**U printf**

左侧第一列表示符号定义在文件中的位置（地址），对程序而言符号就是地址值。T表示符号定义在.text节，U表示未定义符号，即需要从外部获取符号值（地址值）的符号。

ELF文件内通过3个节来实现符号表机制，它们是.symtab、.strtab和.hash。.strtab节保存了符号字符串，可视为符号字符串数组，数组项中以0字节分隔。.symtab节保存符号表数据结构实例，每个符号由固定大小的数据结构实例表示，结构中包含符号值（地址值）、类型等信息，而符号名称字符串由.strtab节中字符串索引值表示，也就是说字符串保存在.strtab节，.symtab节中只保存字符串在.strtab节内的索引值。.hash节保存散列表，用于帮助索引值到符号字符串的快速查找。

在内核代码中，能被其它模块引用的符号（变量和函数）需要将其导出。在加载模块解决引用时需要扫描模块目标文件中的符号表，对未定义的符号，扫描内核和其依赖模块的导出符号列表，获取符号地址解决引用。 内核（模块）为了导出符号，在可执行文件（目标文件中）增加了额外的节，用于收集导出符号信息。

内核提供了声明导出符号的宏：EXPORT\_SYMBOL和EXPORT\_SYMBOL\_GPL等。二者分别用于导出符号和导出只用于GPL兼容代码的符号，它们的目的是将相应的符号信息放置到目标文件指定节当中。

以上两个宏定义在/include/linux/export.h头文件内：

#define EXPORT\_SYMBOL(sym) \ /\*sym表示变量名或函数名\*/

\_\_EXPORT\_SYMBOL(sym, "")

#define EXPORT\_SYMBOL\_GPL(sym) \

\_\_EXPORT\_SYMBOL(sym, "\_gpl")

#define EXPORT\_SYMBOL\_GPL\_FUTURE(sym) \

\_\_EXPORT\_SYMBOL(sym, "\_gpl\_future")

#define **\_\_EXPORT\_SYMBOL**(sym, sec) \

extern typeof(sym) sym; \

\_\_CRC\_SYMBOL(sym, sec) \ /\*符号校验和，保存在指定节\*/

static const char **\_\_kstrtab\_##sym[]** \ /\*定义字符串，保存在指定节中\*/

\_\_attribute\_\_((section("**\_\_ksymtab\_strings**"), aligned(1))) \

= **MODULE\_SYMBOL\_PREFIX #sym**; \

static const struct **kernel\_symbol \_\_ksymtab\_##sym** \ /\*定义并初始化kernel\_symbol实例\*/

\_\_used \

\_\_attribute\_\_((section("**\_\_\_ksymtab" sec "+" #sym**), unused)) \ /\*保存在指定节中\*/

= { **(unsigned long)&sym**, **\_\_kstrtab\_##sym** } /\*‘##’表示字符串拼接\*/

\_\_EXPORT\_SYMBOL(sym, sec)宏的主要功能有（还导出检验和保存在指定节）：

（1）定义名称为\_kstrtab\_*sym*的字符数组，其内容为MODULE\_SYMBOL\_PREFIX #sym，并保存在指定\_\_ksymtab\_strings节中。

（2）定义kernel\_symbol结构体实例保存在\_\_\_ksymtab*sec+sym*节中。

简单地说就是定义一个表示名称的字符数组位于保存到指定节，定义一个kernel\_symbol结构体实例保存到指定节。

kernel\_symbol结构体定义在/include/linux/export.h头文件：

struct kernel\_symbol

{

unsigned long value; /\*符号值，地址值\*/

const char \*name; /\*符号名称字符串指针，指向\_kstrtab\_***sym***字符数组\*/

};

kernel\_symbol结构体包含两个成员，分别表示符号的值（地址值）以及符号名称字符串指针。在宏定义中value成员赋值为符号地址值，name成员赋值为定义字符数组地址。下图示意了宏定义创建的数据结构实例关系：



在内核和模块代码都会使用以上两个宏，因此在内核可执行文件和模块目标文件内会生成以下节：

**●**\_\_\_ksymtab+\*：保存EXPORT\_SYMBOL(sym)宏中创建的kernel\_symbol实例。

**●**\_\_\_ksymtabgpl+\*：保存EXPORT\_SYMBOL\_GPL(sym)宏中创建的kernel\_symbol实例。

**●**\_\_\_ksymtabgpl\_future+\*：保存EXPORT\_SYMBOL\_GPL\_FUTURE(sym)宏中创建的kernel\_symbol实例。

**●**\_\_ksymtab\_strings：保存所有导出符号名称字符串。

内核在/include/asm-generic/vmlinux.lds.h头文件中定义了标识只读数据段的宏：

#define RO\_DATA\_SECTION(align) \

. = ALIGN((align)); \

... \

\_\_ksymtab : AT(ADDR(\_\_ksymtab) - LOAD\_OFFSET) { \

VMLINUX\_SYMBOL(\_\_start\_\_\_ksymtab) = .; \ /\*节起始地址\*/

\*(SORT(**\_\_\_ksymtab+\***)) \ /\*\_\_\_ksymtab+\*节\*/

VMLINUX\_SYMBOL(\_\_stop\_\_\_ksymtab) = .; \ /\*节结束地址，下同\*/

} \

\

\_\_ksymtab\_gpl : AT(ADDR(\_\_ksymtab\_gpl) - LOAD\_OFFSET) { \

VMLINUX\_SYMBOL(\_\_start\_\_\_ksymtab\_gpl) = .; \

\*(SORT(**\_\_\_ksymtab\_gpl+\***)) \ /\*\_\_\_ksymtab\_gpl+\*节\*/

VMLINUX\_SYMBOL(\_\_stop\_\_\_ksymtab\_gpl) = .; \

} \

...

\_\_ksymtab\_gpl\_future : AT(ADDR(\_\_ksymtab\_gpl\_future) - LOAD\_OFFSET) { \

VMLINUX\_SYMBOL(\_\_start\_\_\_ksymtab\_gpl\_future) = .; \

\*(SORT(**\_\_\_ksymtab\_gpl\_future+\***)) \ /\*\_\_\_ksymtab\_gpl\_future+\*节\*/

VMLINUX\_SYMBOL(\_\_stop\_\_\_ksymtab\_gpl\_future) = .; \

} \

... \ /\*保存校验和的节\*/

\

/\*\_\_ksymtab\_strings节\*/ \

\_\_ksymtab\_strings : AT(ADDR(\_\_ksymtab\_strings) - LOAD\_OFFSET) { \

\*(**\_\_ksymtab\_strings**) \

} \

... \

. = ALIGN((align));

#define RODATA RO\_DATA\_SECTION(4096) /\*嵌入链接脚本\*/

Linux系统在/proc/kallsyms文件中列出了导出符号的地址和名称（只读）。

模块链接文件为/scripts/module-common.lds，如下所示：

SECTIONS {

/DISCARD/ : { \*(.discard) }

\_\_ksymtab 0 : { \*(SORT(\_\_\_ksymtab+\*)) }

\_\_ksymtab\_gpl 0 : { \*(SORT(\_\_\_ksymtab\_gpl+\*)) }

\_\_ksymtab\_unused 0 : { \*(SORT(\_\_\_ksymtab\_unused+\*)) }

\_\_ksymtab\_unused\_gpl 0 : { \*(SORT(\_\_\_ksymtab\_unused\_gpl+\*)) }

\_\_ksymtab\_gpl\_future 0 : { \*(SORT(\_\_\_ksymtab\_gpl\_future+\*)) }

\_\_kcrctab 0 : { \*(SORT(\_\_\_kcrctab+\*)) }

\_\_kcrctab\_gpl 0 : { \*(SORT(\_\_\_kcrctab\_gpl+\*)) }

\_\_kcrctab\_unused 0 : { \*(SORT(\_\_\_kcrctab\_unused+\*)) }

\_\_kcrctab\_unused\_gpl 0 : { \*(SORT(\_\_\_kcrctab\_unused\_gpl+\*)) }

\_\_kcrctab\_gpl\_future 0 : { \*(SORT(\_\_\_kcrctab\_gpl\_future+\*)) }

. = ALIGN(8);

.init\_array 0 : { \*(SORT(.init\_array.\*)) \*(.init\_array) }

}

下图简单示意了模块加载后，解决符号引用的过程。模块被加载到内核地址空间VMALLOC区，扫描其符号表中的未决符号，搜索内核导出符号列表，获取符号地址，解决符号引用。如果模块引用了其它模块中的导出符号，还将搜索其它模块的导出符号列表。



### 8.10.2生成模块

模块由内核中源代码文件生成。在配置内核时，可选择将某个子系统、驱动程序等代码永久编译入内核（即直接链接到内核可执行文件）或生成为模块。模块是可重定位的目标文件，模块独立于内核可执行文件，在运行时加载到内存中后再与内核链接，这类似于用户空间的动态库。模块作为特殊的目标文件，增加了一些额外的节，表示模块的信息。

#### 1额外信息

模块目标文件内除了增加导出符号的节之外，还有其它的节。例如：表示模块参数的节、表示模块信息的节以及在内核中表示模块数据结构的节等。

模块的额外参数通过宏来定义。模块源代码可永久编译入内核，也可编译成模块。永久编译入内核时这些宏定义有不同的实现（更为简单），内核在将源文件编译成模块时，在编译选项中会添加MODULE宏的定义，也就是说在将源代码永久编译入内核时，不会定义MODULE宏，编译成模块时会定义MODULE宏，详见/Makefile文件中对KBUILD\_AFLAGS\_MODULE和KBUILD\_CFLAGS\_MODULE的定义。

##### ■**模块参数**

模块参数是模块内的全局变量，可以在加载模块时对其赋值，形式如“参数名称=值”，这与引导加载程序传递命令行参数十分相似。模块中声明模块参数的宏如下（/include/linux/moduleparam.h）：

#define **module\_param**(name, type, perm) \

module\_param\_named(name, name, type, perm) /\*/include/linux/moduleparam.h\*/

name：表示模数的名称，

type：表示变量类型，如byte, short, ushort, int, uint, long, ulong,charp,bool等。

perm：表示访问权限，与文件权限控制相同。

module\_param\_named()宏定义如下：

#define module\_param\_named(name, value, type, perm) \

param\_check\_##type(name, &(value)); \ /\*/include/linux/moduleparam.h\*/

**module\_param\_cb**(name, &param\_ops\_##type, &value, perm); \ /\*定义kernel\_param实例\*/

\_\_MODULE\_PARM\_TYPE(name, #type) /\*在.modinfo节中导出参数信息\*/

module\_param\_named()宏的最终效果是创建并设置kernel\_param结构体实例，并将实例链接到名称为"\_\_param"的节，并将参数信息导出到.modinfo节，这些节都是位于模块目标文件中的节。

kernel\_param实例关联的param\_ops\_##type实例为kernel\_param\_ops结构体实例，根据变量类型的不同，实例也不相同，实例中主要包含读写变量值的函数，详见第2章（/kernel/params.c）。

在加载模块时，系统调用会将传递的模块参数与"\_\_param"节内的kernel\_param实例比较，相同则调用关联kernel\_param\_ops实例中的设置函数将变量值写入模块中变量。

##### **■模块信息**

模块还有许多其它信息通常通过MODULE\_INFO(tag, info)宏定义（/include/linux/module.h）：

#define MODULE\_INFO(tag, info) \_\_MODULE\_INFO(tag, tag, info)

\_\_MODULE\_INFO()宏定义在/include/linux/moduleparam.h头文件：

#define \_\_MODULE\_INFO(tag, name, info) \

static const char \_\_UNIQUE\_ID(name)[] \ /\*定义字符数组\*/

\_\_used \_\_attribute\_\_((section("**.modinfo**"), unused, aligned(1))) \ /\*保存在.modinfo节内\*/

= \_\_stringify(tag) "=" info /\*字符数组内容\*/

\_\_MODULE\_INFO()宏的功能就是定义字符数组，内容为“tag=info”，并将其保存到**.modinfo**节内。

在/include/linux/module.h头文件中定义了声明模块信息宏，例如：

**MODULE\_SOFTDEP**(\_softdep)：必须在本模块之前加载的模块。

MODULE\_ALIAS(\_alias)：模块别名（用户空间可用），“alias=*\_alias*”。

MODULE\_LICENSE(\_license)：模块版权，“license=*\_license*”。

MODULE\_AUTHOR(\_author)：模块作者，“author=*\_author*”。

MODULE\_DESCRIPTION(\_description)：模块描述，“description=*\_description*”。

##### ■**加/卸载函数**

加/卸载函数是指在模块被加/卸载时调用的模块内部函数。

声明模块加载和卸载函数的宏分别为**module\_init(initfn)**和**module\_exit(exitfn)。**

这两个宏定义在/include/linux/module.h头文件：

#define module\_init(initfn) \

static inline initcall\_t \_\_inittest(void) \

{ return initfn; } \

int **init\_module**(void) \_\_attribute\_\_((alias(#initfn))); /\*initfn函数取别名init\_module\*/

#define module\_exit(exitfn) \

static inline exitcall\_t \_\_exittest(void) \

{ return exitfn; } \

void **cleanup\_module**(void) \_\_attribute\_\_((alias(#exitfn))); /\*exitfn函数取别名cleanup\_module\*/

initfn为加载函数名称，exitfn为卸载函数名称。module\_init()和module\_exit()宏分别为这两个函数取了别名init\_module和cleanup\_module，这两个别名在所有模块中都是一样的。

#### 2编译模块

生成模块需要执行以下3个步骤：

（1）将模块包含的所有源文件（.c）编译成普通的.o目标文件。

（2）生成目标文件后，分析模块，找到附加信息（如模块依赖关系），保存到一个独立的附加文件。

（3）将普通目标文件与附加文件链接，生成模块（.ko）。

生成第（2）步附加文件的程序（脚本）位于/scripts/mod/目录下，/scripts/mod/modpost.c文件内的main()函数用于向附加文件写入内容。

在第（2）步生成的附加文件中，在.gun.linkonce.module节保存了module结构体实例，实例中保存了模块信息，内核可直接使用，后面将详细介绍。

在.modinfo节将写入"**depends=\*\*\***"模块信息，其值是本模块依赖的模块名称，在.modinfo节中还将写入其它模块信息，这里的.modinfo节将与第（1）步生成的目标文件中的.modinfo节合并。

在附加文件中还将创建其它的节，用于保存模块信息，这里就不一一介绍了（作者能力有限！）。

在执行make命令构建内核时将构建内核模块，而后也可以使用**make modules**重新编译所有的模块。

模块也可以单独编译，但需要已构建好的内核源码树，单独编译模块的命令如下：

**make -C $KDIR M=$PWD**

$KDIR表示内核源代码所在目录（绝对路径），$PWD表示模块代码所在目录（绝对路径），执行此命令后，Kbuild机制将读取模块代码所在目录下的Makefile文件，编译模块。

在Makefile文件中，obj-m列表下的目标文件将编译成模块，模块目标文件后缀为.ko。如果模块由单个文件组成，则Makefile文件内容如下（drivers/isdn/i4l/Makefile）：

obj-$(CONFIG\_ISDN\_PPP\_BSDCOMP) += isdn\_bsdcomp.o /\*配置选项值为m\*/

如果模块由多个文件组成，则Makefile文件格式如下（/drivers/isdn/i4l/Makefile）：

obj-$(CONFIG\_ISDN\_I4L) += isdn.o

isdn-y := isdn\_net\_lib.o isdn\_v110.o isdn\_common.o

isdn.o表示模块名称，在构建内核时Kbuild将编译、链接isdn-y内包含的目标文件，以生成模块。

内核、模块更详细的配置、构建信息请参考第14章。

### 8.10.3加载模块

内核中通过module结构体管理模块，每个加载的模块在内核中一个实例表示。用户进程可通过系统调用向内核添加模块，也可通过用户空间工具添加模块。内核添加模块的方式也是通过激活用户进程实现的。

#### 1管理模块

在内核中每个加载的模块由module结构体表示。在构建模块时，在自动生成的附加文件中包含此结构体实例并初始化，保存在.gnu.linkonce.this\_module节（/scripts/mod/modpost.c），内核可直接使用此实例。

module结构体定义在/include/linux/module.h头文件，其成员简列如下：

struct module {

enum module\_state **state**; /\*模块状态，格举类型\*/

struct list\_head **list**; /\*链表成员，将实例链接到全局管理链表\*/

char **name[MODULE\_NAME\_LEN]**; /\*模块名称，用于查找模块\*/

... /\*导出sysfs相关成员\*/

const struct kernel\_symbol \***syms**; /\*模块导出符号列表\*/

const unsigned long \*crcs;

unsigned int num\_syms;

/\*内核参数\*/

#ifdef CONFIG\_SYSFS

struct mutex param\_lock;

#endif

struct kernel\_param \*kp; /\*模块参数列表\*/

unsigned int num\_kp;

/\*GPL兼容导出符号\*/

unsigned int num\_gpl\_syms;

const struct kernel\_symbol \*gpl\_syms;

const unsigned long \*gpl\_crcs;

... /\*未使用符号\*/

bool async\_probe\_requested;

/\*将来只兼容GPL导出符号\*/

const struct kernel\_symbol \*gpl\_future\_syms;

const unsigned long \*gpl\_future\_crcs;

unsigned int num\_gpl\_future\_syms;

/\* Exception table \*/

unsigned int num\_exentries;

struct exception\_table\_entry \*extable;

**int (\*init)(void)**; /\*加载函数指针，赋予**init\_module()**指针\*/

void \*module\_init \_\_\_\_cacheline\_aligned;

void \*module\_core;

unsigned int init\_size, core\_size;

unsigned int init\_text\_size, core\_text\_size;

...

char \***args**; /\*加载模块时的命令行参数指针\*/

...

#ifdef CONFIG\_MODULE\_UNLOAD /\*模块可卸载\*/

struct list\_head **source\_list**; /\*依赖本模块的模块链表\*/

struct list\_head **target\_list**; /\*本模块依赖的模块链表\*/

void **(\*exit)(void);**  /\*卸载函数指针，赋予**cleanup\_module()**指针\*/

atomic\_t refcnt;

#endif

...

} \_\_\_\_cacheline\_aligned;

module结构体部分成员简介如下：

●**state：**模块状态，枚举类型module\_state定义如下（/include/linux/module.h）：

enum module\_state {

MODULE\_STATE\_LIVE, /\*正常状态\*/

MODULE\_STATE\_COMING, /\*装载状态\*/

MODULE\_STATE\_GOING, /\* Going away. \*/

MODULE\_STATE\_UNFORMED, /\* Still setting it up. \*/

};

**●list：**内核定义了全局链表modules（/kernel/module.c），用于管理内核所有的module实例，list为双链表成员。

**●name：**模块名称名符数组，必须是唯一的，内核由名称来标识模块，一般用模块二进制文件名去掉.ko后缀用于该成员。

●**init**：模块加载时调用的初始化函数指针，赋值为init\_module()函数指针。

●**exit**：模块卸载时调用的函数指针，赋值为cleanup\_module()函数指针。

●**syms、gpl\_syms、gpl\_future\_syms：**模块导出符号的kernel\_symbol结构体列表。

●**source\_list，target\_list：**双链表成员。模块内部可能调用其它模块内定义的函数或变量，因此模块之间会形成依赖关系。例如：模块A内调用了模块B内的函数，则A依赖B，在装载A之前需先装载B，在卸载B之前必须保证依赖它的模块A已经卸装。

以上两个双链表成员链接的是module\_use结构体实例（/include/linux/module.h）：

struct module\_use {

struct list\_head source\_list; /\*依赖本模块的模块链表\*/

struct list\_head target\_list; /\*本模块依赖的模块链表\*/

**struct module \*source, \*target**; /\*模块指针\*/

};

source\_list成员用于链接依赖本模块的模块，如以上例子中模块A位于B模块此链表内，target\_list表示本模块依赖的模块链表，以上例子中模块B位于模块A此链表内，如下图所示。



加载的模块信息还将通过sysfs文件系统导到/proc/modules文件内。

#### 2系统调用

模块通常安装到系统/lib/modules/$(KERNELRELEASE)/目录下， $(KERNELRELEASE)表示内核版本号，如4.2.4（目录名称）。

内核和用户都可以发起模块的加载（需要特权），最终加载操作都通过init\_module()/finit\_module()系统调用实现。这里先简要介绍init\_module()系统调用的实现，后面再介绍内核和用户如何发起模块的加载。

在调用init\_module()系统调用前，用户进程需打开模块（.ko文件），并将模块内容起始地址及长度传递给系统调用。

init\_module()系统调用实现代码位于/kernel/module.c文件内，代码简列如下：

SYSCALL\_DEFINE3(init\_module, void \_\_user \*, umod,unsigned long, len, const char \_\_user \*, uargs)

/\*umod：指向用户空间映射模块文件内容的基地址（指向文件内容开头）

\*len：模块长度，uargs：参数。

\*/

{

int err;

struct **load\_info** info = { }; /\*暂存加载信息，/kernel/module.c\*/

err = may\_init\_module(); /\*需CAP\_SYS\_MODULE能力，且内核允许加载模块\*/

...

err = copy\_module\_from\_user(umod, len, &info); /\*复制模块文件内容至内核VMALLOC区\*/

...

return **load\_module**(&info, uargs, 0); /\*解决引用，调用模块初始化（加载）函数等\*/

}

init\_module()系统调用内将模块文件内容复制到内核VMALLOC区，然后调用load\_module()函数解决引用，调用模块初始化函数等。

load\_module()函数代码简列如下（/kernel/module.c）：

static int load\_module(struct load\_info \*info, const char \_\_user \*uargs,int flags)

{

struct module \*mod;

long err;

char \*after\_dashes;

... /\*在效性检查\*/

/\* Figure out module layout, and allocate all the memory. \*/

mod = layout\_and\_allocate(info, flags); /\*获取模块文件更详细信息，如各节信息等\*/

...

/\* Reserve our place in the list. \*/

err = add\_unformed\_module(mod); /\*mod指向module实例，添加到内核管理结构等\*/

...

/\* To avoid stressing percpu allocator, do this once we're unique. \*/

err = percpu\_modalloc(mod, info); /\*SMP，分配保留内存\*/

...

/\* Now module is in final location, initialize linked lists, etc. \*/

err = module\_unload\_init(mod); /\*初始化依赖链表\*/

...

init\_param\_lock(mod);

err = **find\_module\_sections**(mod, info); /\*获取各导出符号节等信息，写入module实例\*/

...

err = check\_module\_license\_and\_versions(mod);

...

/\* Set up MODINFO\_ATTR fields \*/

setup\_modinfo(mod, info); /\*处理模块属性\*/

/\* Fix up syms, so that st\_value is a pointer to location. \*/

err = **simplify\_symbols**(mod, info); /\*解决未定义符号引用等\*/

...

err = **apply\_relocations**(mod, info); /\*模块内符号重定位\*/

...

err = post\_relocation(mod, info); /\*处理内核导出符号，体系结构相关工作等\*/

...

flush\_module\_icache(mod);

/\*复制命令行参数\*/

mod->args = strndup\_user(uargs, ~0UL >> 1);

...

dynamic\_debug\_setup(info->debug, info->num\_debug);

/\* Ftrace init must be called in the MODULE\_STATE\_UNFORMED state \*/

ftrace\_module\_init(mod);

/\* Finally it's fully formed, ready to start executing. \*/

err = complete\_formation(mod, info); /\*收尾工作\*/

...

/\* Module is ready to execute: parsing args may do that. \*/

after\_dashes = parse\_args(mod->name, mod->args, mod->kp, mod->num\_kp,

-32768, 32767, NULL,unknown\_module\_param\_cb); /\*处理命令行参数\*/

...

/\*导出信息至sysfs\*/

err = mod\_sysfs\_setup(mod, info, mod->kp, mod->num\_kp);

...

trace\_module\_load(mod);

return **do\_init\_module**(mod); /\*调用模块初始化函数**mod->init()**等\*/

...

}

加载模块操作简单地说就是将模块文件内容复制到内核VMALLOC区，设置表示模块的module实例，添加到内核管理结构，解决未定义符号引用和模块自身的重定位，最后调用模块初始化函数。

在解决未定义符号引用时，对于引用的内核导出符号，搜索内核导出符号列表解决引用，对于引用其它模块导出的符号，则搜索内核modules链表中的module实例，实例中包括模块导出符号，从而解决引用。

init\_module()系统调用需要用户进程先打开模块文件，映射到用户空间，然后将映射文件内容基址作为参数，调用init\_module()系统调用。

finit\_module()系统调用也可以用来加载模块，且只需要先打开模块文件获取文件描述符即可，将文件描述符传递给系统调用，而不需要将文件内容映射到用户空间，它直接复制文件内容到内核VMALLOC区，然后调用load\_module()函数完成加载操作，实现函数请读者自行阅读。

移除模块的系统调用为delete\_module()，请读者自行阅读实现代码。

#### 3加载操作

用户和内核都可以发起模块的加载。

从用户角度可通过系统工具insmod和modprobe命令加载模块，insmod用于加载单一的模块，不考虑依赖关系，系统调用内先将模块文件内容映射到用户空间，或打开模块文件，然后调用init\_module()或finit\_module()系统调用完成模块加载。

modprobe会分析模块之间的依赖关系，在识别依赖关系后调用init\_module()系统调用依次加载模块。

内核态模块的加载由**request\_module(mod...)**函数（/include/linux/kmod.h）发起，参数mod表示模块名称，函数内调用/kernel/kmod.c内的\_\_request\_module()函数，函数内启动用户工具"/sbin/modprobe"完成模块的加载。

在向内核注册device实例时，将向用户空间发送uevent事件信息，用户工具“/sbin/udevd”（守护进程）接收uevent事件信息，然后加载以模块编译的设备驱动程序（代替"/sbin/modprobe"）。

## 8.11小结

本章介绍了内核用来管理设备和设备驱动程序的通用驱动模型。通用驱动模型中主要包括总线、设备类、设备、驱动等概念。

总线用于管理挂接在其上的设备和驱动，设备类用来管理同一类型的设备。设备和驱动挂接到某一总线上，在向总线注册设备或驱动时，会查找总线上匹配的驱动或设备，调用总线定义的匹配函数，若匹配成功将调用总线或驱动中的探测函数，在探测函数中将加载设备的驱动程序。

本章还介绍了platform、SPI、I2C、USB等具体总线（主机控制器）及设备驱动的实现，以及设备树和模块等相关内容。